

---

# **Rocprofiler SDK**

*Release 1.1.0*

**Advanced Micro Devices, Inc.**

**May 25, 2026**



# INSTALL

<b>1</b>	<b>Installing ROCprofiler-SDK</b>	<b>3</b>
1.1	Supported systems	3
1.2	Identifying the operating system	3
1.3	Build requirements	3
1.3.1	Install CMake	4
1.4	Building ROCprofiler-SDK from source	4
1.5	Installing ROCprofiler-SDK	5
1.6	Testing ROCprofiler-SDK	5
1.7	Install using package manager	5
<b>2</b>	<b>ROCprofiler-SDK Quick Reference Guide</b>	<b>7</b>
2.1	Getting Started	7
2.2	Essential Commands	7
2.2.1	Querying System Capabilities	7
2.2.2	Basic Tracing	7
2.2.3	Granular Tracing Options	8
2.2.4	Performance Counter Collection	8
2.3	Advanced Profiling Features	8
2.3.1	PC Sampling (Beta)	8
2.3.2	Thread Trace	8
2.3.3	Process Attachment	8
2.4	Output Formats and Post-processing	9
2.4.1	Working with rocpd Database Format	9
2.4.2	Collecting and converting to Other Formats	9
2.4.3	Summary and Statistics	9
2.5	Filtering and Selection	9
2.5.1	Kernel Filtering	9
2.5.2	Time-based Collection	9
2.6	Kernel Naming and Display	10
2.7	Code Annotation with ROCTx	10
2.8	Parallel and Distributed Applications	10
2.8.1	MPI Applications	10
2.8.2	OpenMP Applications	10
2.9	Output Management	10
2.9.1	File Organization	10
2.10	Common Use Cases	11
2.10.1	Basic Performance Analysis	11
2.10.2	Detailed Kernel Analysis	11
2.10.3	Memory Transfer Analysis	11
2.10.4	Timeline Visualization	11

2.11	Installation and Setup	11
2.12	Troubleshooting Quick Tips	11
<b>3</b>	<b>ROCprofiler-SDK samples</b>	<b>13</b>
3.1	Finding samples	13
3.2	Building Samples	13
3.3	Running samples	13
<b>4</b>	<b>Using rocprofv3</b>	<b>15</b>
4.1	Command-line options	15
4.2	Application tracing	17
4.2.1	HIP trace	17
4.2.2	HSA trace	21
4.2.3	Marker trace	23
4.2.4	Kokkos trace	23
4.2.5	Kernel trace	23
4.2.6	Memory copy trace	25
4.2.7	Memory allocation trace	25
4.2.8	Runtime trace	26
4.2.9	System trace	26
4.2.10	Scratch memory trace	26
4.2.11	RCCL trace	27
4.2.12	rocDecode trace	30
4.2.13	rocJPEG trace	30
4.2.14	Process Attachment	31
4.2.15	Post-processing tracing options	31
4.2.15.1	Stats	31
4.2.15.2	Summary	33
4.2.15.3	Summary per domain	33
4.2.15.4	Summary groups	33
4.2.15.5	Summary output file	34
4.2.16	Configuration Output	42
4.2.17	Collecting traces using input file	44
4.2.18	Disabling specific tracing options	45
4.3	Kernel counter collection	46
4.3.1	Counter collection using input file	46
4.3.2	Counter collection using command line	48
4.3.3	Extra counters	48
4.3.4	Kernel counter collection output	49
4.3.5	Iteration based counter multiplexing	50
4.4	Perfetto visualization	51
4.4.1	Perfetto visualization for traces	51
4.4.2	Perfetto visualization for counter collection	52
4.4.3	Scratch Memory Visualization in Perfetto	53
4.5	Agent info	54
4.6	Advanced options	54
4.6.1	Minimum output data threshold	54
4.6.2	Signal handler control	56
4.6.3	Library preloading	58
4.6.4	ROCm root path configuration	60
4.6.5	SDK shared object version control	61
4.6.6	SDK version specification	62
4.6.7	Agent index	64
4.6.8	Group by queue	65

4.7	Kernel naming and filtering . . . . .	66
4.7.1	Kernel name mangling . . . . .	66
4.7.2	Kernel name truncation . . . . .	67
4.7.3	Kernel filtering . . . . .	67
4.7.4	Kernel rename . . . . .	68
4.8	I/O control options . . . . .	69
4.8.1	Output prefix keys . . . . .	69
4.8.2	Output directory . . . . .	69
4.8.3	Output file . . . . .	70
4.8.4	Collection period . . . . .	70
4.8.5	Perfetto-specific options . . . . .	71
4.9	Output file fields . . . . .	71
4.10	Output formats . . . . .	72
4.10.1	JSON output schema . . . . .	73
4.10.1.1	Properties . . . . .	73
<b>5</b>	<b>Using rocprofv3-avail</b>	<b>83</b>
5.1	Command-line options . . . . .	83
<b>6</b>	<b>Dynamic process attachment using rocprofv3</b>	<b>85</b>
6.1	Basic dynamic process attachment . . . . .	85
6.2	Duration-specific dynamic process attachment . . . . .	86
6.3	Dynamic process attachment with counter collection . . . . .	86
6.4	Key considerations . . . . .	86
<b>7</b>	<b>Using rocpd output format</b>	<b>87</b>
7.1	Features . . . . .	87
7.2	Generating rocpd output . . . . .	87
7.3	Converting rocpd to alternative formats . . . . .	88
7.3.1	rocpd convert command-line options . . . . .	88
7.4	Types of conversion . . . . .	90
7.5	Dedicated conversion tools . . . . .	90
7.5.1	rocpd2csv - CSV export tool . . . . .	90
7.5.2	rocpd2otf2 - OTF2 export tool . . . . .	91
7.5.3	rocpd2pftrace - Perfetto trace export . . . . .	92
7.5.4	rocpd2summary - statistical analysis tool . . . . .	93
7.6	Generating performance summary using rocpd . . . . .	94
7.6.1	Multidatabase summary analysis . . . . .	95
7.6.1.1	Use cases for multidatabase summary analysis . . . . .	95
7.6.2	Integration with rocprofv3 workflow . . . . .	97
7.7	Aggregating multiprofiling data using rocpd . . . . .	97
7.8	Tool integration and workflow examples . . . . .	98
7.9	Executing SQL queries with rocpd . . . . .	99
7.9.1	Database schema and views . . . . .	100
7.9.2	rocpd query examples . . . . .	100
7.9.3	Output format options . . . . .	102
7.9.4	Script-based analysis . . . . .	102
7.9.5	Time window integration . . . . .	103
7.9.6	Automated Email reporting . . . . .	103
7.9.7	Integration workflows . . . . .	104
7.9.8	rocpd query command-line reference . . . . .	105
<b>8</b>	<b>Using ROCTx</b>	<b>107</b>
8.1	ROCTx annotations . . . . .	107
8.1.1	Markers . . . . .	107

8.1.2	Ranges	107
8.1.3	ROCTx APIs	107
8.2	Using ROCTx in the application	108
8.3	Resource naming	110
8.4	Using ROCTx in the python application	111
<b>9</b>	<b>Using rocprofv3 with MPI</b>	<b>113</b>
9.1	ROCTx annotations	114
9.2	Output format features	116
<b>10</b>	<b>Using rocprofv3 with OpenMP</b>	<b>117</b>
10.1	Example: Vector Addition Using OpenMP Offload on AMD GPUs	117
10.2	Building the OpenMP Offload Application	118
10.3	Profiling the Application with rocprofv3	118
<b>11</b>	<b>Using PC sampling</b>	<b>119</b>
11.1	PC sampling availability and configuration	119
11.2	PC sampling fields	125
11.3	Host-trap PC sampling and arbitrary sampling skid	126
11.4	Hardware-based (stochastic) PC sampling method	127
<b>12</b>	<b>Using thread trace</b>	<b>141</b>
12.1	Prerequisites	141
12.2	rocprofv3 parameters for thread tracing	142
12.3	Using input file	144
12.4	Thread tracing for multiple kernel instances	144
12.5	rocprofv3 output files	144
12.5.1	Stats CSV	145
12.6	Troubleshooting	145
<b>13</b>	<b>ROCprofiler-SDK tool library</b>	<b>147</b>
13.1	ROCm runtimes design	147
13.2	Tool library design	147
13.3	Tool finalization	149
13.4	Full rocprofiler_configure sample	150
<b>14</b>	<b>Runtime intercept tables</b>	<b>153</b>
14.1	Forward declaration of public C API function	153
14.2	Internal implementation of API function	153
14.3	Dispatch table implementation	153
14.4	Implementation of public C API function	154
14.5	Dispatch table chaining	154
<b>15</b>	<b>Implementing Process Attachment Tools</b>	<b>155</b>
15.1	Overview	155
15.2	Python Functions	155
15.3	C Functions	156
15.4	Function Call Sequence	156
15.4.1	Initial Attachment Sequence	156
15.4.2	Reattachment Sequence	156
15.5	Environment Variable Configuration	157
15.5.1	Required Variables	157
15.5.2	Tool Library Configuration	157
15.6	Using the Attachment Functions	157
15.6.1	Basic Attachment Tool Implementation	157

15.6.2	Complete Tool Example	159
<b>16</b>	<b>ROCprofiler-SDK buffered services</b>	<b>161</b>
16.1	Subscribing to buffer tracing services	161
16.2	Creating a buffer	161
16.3	Creating a dedicated thread for buffer callbacks	162
16.4	Configuring buffer tracing services	162
16.5	Buffer tracing callback function	164
16.6	Buffer tracing record	165
<b>17</b>	<b>ROCprofiler-SDK callback tracing services</b>	<b>167</b>
17.1	Subscribing to callback tracing services	167
17.2	Callback tracing callback function	168
17.3	Callback tracing record	170
17.4	Code object tracing	172
<b>18</b>	<b>ROCprofiler-SDK counter collection services</b>	<b>175</b>
18.1	Definitions	175
18.2	Using the Counter Collection Service	176
18.2.1	tool_init() setup	176
18.3	Profile Setup	176
18.4	Dispatch Counting Callback	178
18.5	Agent Set Profile Callback	179
18.5.1	Buffered callback	179
18.6	Counter Definitions	180
18.7	Derived Metrics	180
18.7.1	Reduce Function	180
18.7.2	Select Function	181
18.8	Accumulate Function	182
18.9	Kernel Serialization	183
<b>19</b>	<b>ROCprofiler-SDK PC sampling method</b>	<b>185</b>
19.1	ROCprofiler-SDK PC sampling service	185
19.1.1	tool_init() setup	185
19.2	Processing PC samples	187
<b>20</b>	<b>ROCprof Trace Decoder and thread trace APIs</b>	<b>189</b>
20.1	Thread trace service API	189
20.1.1	tool_init() setup	189
20.1.2	Device thread trace	191
20.1.3	Dispatch thread trace	191
20.2	ROCprof Trace Decoder API	192
20.2.1	Trace Decoder setup	192
20.2.2	Code object tracking	193
20.3	Processing thread trace data	193
20.3.1	Decoder callback	194
20.3.2	Trace Decoder info events	194
<b>21</b>	<b>ROCprofiler-SDK API library</b>	<b>197</b>
21.1	Modules	197
21.1.1	Agent Information	197
21.1.2	Buffer handling	199
21.1.3	Buffer tracing	200
21.1.4	Callback tracing	205
21.1.5	Context management	211

21.1.6	Counter config	211
21.1.7	Counters	212
21.1.8	Device counting service	214
21.1.9	Dispatch counting service	216
21.1.10	External correlation	218
21.1.11	Intercept table	222
21.1.12	Internal threading management	223
21.1.13	OMPT Registration	225
21.1.14	PC Sampling service	225
21.1.15	Thread trace	232
21.1.16	Tool registration	240
21.2	Global Data structures, topics, files	244
21.2.1	Global Basic Data Types	244
21.2.2	Topics	267
21.2.3	Data Structures	267
21.2.4	File List	267
<b>22</b>	<b>ROCTx API</b>	<b>269</b>
22.1	Introduction	269
22.2	Modules	270
22.2.1	Markers Information	270
22.2.2	Ranges Information	270
22.2.3	Profiler Control Information	271
22.2.4	Naming Information	271
22.3	Global Data structures, topics, files	273
22.3.1	Global Basic Data Types	273
22.3.2	Topics	273
22.3.3	File List	273
<b>23</b>	<b>Comparing ROCprofiler-SDK to other ROCm profiling tools</b>	<b>275</b>
<b>24</b>	<b>Comparing command-line tool options: ROCprofiler(rocprof, rocprofv2) and ROCprofiler-SDK(rocprofv3)</b>	<b>277</b>
<b>25</b>	<b>Timing Difference Between rocprofv3 and rocprofv1/v2</b>	<b>289</b>
<b>26</b>	<b>Default run of rocprofv3 and rocprofv1/v2</b>	<b>291</b>
<b>27</b>	<b>License</b>	<b>293</b>
	<b>Index</b>	<b>295</b>

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.

The code is open source and hosted at <https://github.com/ROCm/rocm-systems/tree/develop/projects/rocprofiler-sdk>.

**Note**

The ROCprofiler-SDK repository for ROCm 7.0 and earlier is located at <https://github.com/ROCm/rocprofiler-sdk>.

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 documentation is structured as follows:

**Install**

- [Installing ROCprofiler-SDK](#)

**Quick Reference**

- [ROCprofiler-SDK Quick Reference Guide](#)

**How to**

- [Samples](#)
- [Using rocprofv3](#)
- [Using rocprofv3-avail](#)
- [Dynamic process attachment using rocprofv3](#)
- [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.

## INSTALLING ROCProfiler-SDK

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

### 1.1 Supported systems

ROCprofiler-SDK is supported on the Linux distributions specified in the [system requirements](#).

### 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

Install the following dependencies:

- Debian/Ubuntu

```
sudo apt install -y libdw-dev libsqlite3-dev
```

- Red Hat Enterprise Linux/Alma Linux/Rocky Linux/Fedora

```
sudo dnf install elfutils elfutils-devel sqlite-devel clang-tools-extra gcc gcc-c++
↪ cmake make openssl-devel
python3 -m pip install --upgrade pip
python3 -m pip install scikit-build
```

- SUSE Linux Enterprise Server

```
sudo zypper install gcc12 gcc12-c++ cmake make python3-devel elfutils sqlite3-devel
↪ libelf-devel libdw-devel
```

(continues on next page)

(continued from previous page)

```
export CXX=/usr/bin/g++-12
export CC=/usr/bin/gcc-12
```

**Note**

The above `export` statements set the compiler environment variables only for the current terminal session. If you open a new terminal or log out, these variables will be unset. To make these settings permanent, add the following lines to your `~/ .bashrc` file:

```
export CXX=/usr/bin/g++-12
export CC=/usr/bin/gcc-12
```

Alternatively, ensure these variables are set before building ROCprofiler-SDK.

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 --no-checkout --filter=blob:none https://github.com/ROCm/rocm-systems.git
cd rocm-systems
git sparse-checkout init --cone
git sparse-checkout set projects/rocprofiler-sdk
git checkout develop
python3 -m pip install -r projects/rocprofiler-sdk/requirements.txt
cmake
  -B rocprofiler-sdk-build
  -DCMAKE_INSTALL_PREFIX=/opt/rocm
  -DCMAKE_PREFIX_PATH=/opt/rocm
  projects/rocprofiler-sdk
cmake --build rocprofiler-sdk-build --target all --parallel $(nproc)
```

## 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
```



## ROCPROFILER-SDK QUICK REFERENCE GUIDE

This quick reference guide provides an overview of the most commonly used `rocprofv3` commands and links to detailed documentation sections.

### 2.1 Getting Started

Export the ROCm binary path:

```
source /opt/rocm/share/rocprofiler-sdk/setup-env.sh
```

Check `rocprofv3` version and help:

```
rocprofv3 --version  
rocprofv3 --help
```

### 2.2 Essential Commands

#### 2.2.1 Querying System Capabilities

List available counters and capabilities:

```
# List all available features  
rocprofv3 --list-avail  
  
# Using the dedicated tool for detailed queries  
rocprofv3-avail list  
rocprofv3-avail info
```

**Documentation:** *Using rocprofv3-avail*

#### 2.2.2 Basic Tracing

Application tracing (HIP API + kernel dispatches + memory operations):

```
# Runtime tracing (recommended for most use cases)  
rocprofv3 --runtime-trace -- ./your_app  
  
# System-level tracing (includes HSA API)  
rocprofv3 --sys-trace -- ./your_app
```

**Documentation:** *Using rocprofv3*

## 2.2.3 Granular Tracing Options

```
# HIP API, kernel dispatches, and memory operations tracing
rocprofv3 --hip-trace --kernel-trace --memory-copy-trace -- ./your_app
```

**Documentation:** *Using rocprofv3* (Basic tracing section)

## 2.2.4 Performance Counter Collection

```
# List available counters
rocprofv3-avail list --pmc

# Check if counters can be collected together
rocprofv3-avail pmc-check SQ_WAVES SQ_INSTS_VALU

# Collect specific counters
rocprofv3 --pmc SQ_WAVES,SQ_INSTS_VALU -- ./your_app
```

**Documentation:** *Using rocprofv3* (Counter collection section)

## 2.3 Advanced Profiling Features

### 2.3.1 PC Sampling (Beta)

```
# Check PC sampling support
rocprofv3-avail list --pc-sampling

# Enable PC sampling
rocprofv3 --pc-sampling-beta-enabled --pc-sampling-interval 1000 -- ./your_app
```

**Documentation:** *Using PC sampling*

### 2.3.2 Thread Trace

```
# Collect thread trace data
rocprofv3 --att --output-format csv -- ./your_app
```

**Documentation:** *Using thread trace*

### 2.3.3 Process Attachment

```
# Attach to a running process by PID
rocprofv3 --pid 12345 --runtime-trace -d ./results
# or

# Attach for a specific duration (10 seconds)
rocprofv3 --pid 12345 --runtime-trace --attach-duration-msec 1000
```

**Documentation:** *using-rocprofv3-process-attachment*

## 2.4 Output Formats and Post-processing

rocprofv3 supports multiple output formats for different analysis needs. The default format is rocpd, which stores data in a structured SQLite3 database.

### 2.4.1 Working with rocpd Database Format

```
# Generate rocpd database (default format)
rocprofv3 --runtime-trace -- ./your_app
# Creates: hostname/pid_results.db

# Query the database directly with SQL
sqlite3 hostname/12345_results.db "SELECT * FROM regions;"

# Convert rocpd database to other formats
rocprof convert -i *.db -f csv pftrace otf2 --start 20% --end 80%
```

### 2.4.2 Collecting and converting to Other Formats

```
# Multiple output formats in one run
rocprofv3 --runtime-trace --output-format csv json pftrace otf2 -- ./your_app
```

**Documentation:** *Using rocpd output format*

### 2.4.3 Summary and Statistics

```
# Overall summary statistics per domain grouped by kernel and memory operations
rocprofv3 --runtime-trace --summary-per-domain --summary-groups "KERNEL_DISPATCH|MEMORY_
↪COPY" -- ./your_app
```

**Documentation:** *Using rocprofv3* (Post-processing tracing section)

## 2.5 Filtering and Selection

### 2.5.1 Kernel Filtering

```
# Include specific kernels by regex
rocprofv3 --kernel-trace --kernel-iteration-range 10-20 --kernel-include-regex "matmul.*
↪" --kernel-exclude-regex ".*copy.*" -- ./your_app
```

**Documentation:** *Using rocprofv3* (Filtering section)

### 2.5.2 Time-based Collection

```
# Collect for specific time periods (start_delay:collection_time:repeat)
rocprofv3 --runtime-trace --collection-period 500:2000:0 --collection-period-unit msec --
↪ ./your_app
```

**Documentation:** *Using rocprofv3* (Filtering section)

## 2.6 Kernel Naming and Display

```
# Keep mangled kernel names
rocprofv3 --kernel-trace --mangled-kernels -- ./your_app

# Truncate kernel names for readability
rocprofv3 --kernel-trace --truncate-kernels -- ./your_app

# Use ROCTx regions to rename kernels
rocprofv3 --kernel-trace --kernel-rename -- ./your_app
```

**Documentation:** *Using rocprofv3* (Kernel naming section)

## 2.7 Code Annotation with ROCTx

```
# Trace ROCTx markers and ranges
rocprofv3 --marker-trace -- ./your_app
```

**Documentation:** *Using ROCTx*

## 2.8 Parallel and Distributed Applications

### 2.8.1 MPI Applications

```
# Profile MPI applications
mpirun -n 4 rocprofv3 --runtime-trace --output-format csv -- ./your_mpi_app
```

**Documentation:** *Using rocprofv3 with MPI*

### 2.8.2 OpenMP Applications

```
# Profile OpenMP applications
rocprofv3 --runtime-trace --output-format csv -- ./your_openmp_app
```

**Documentation:** *Using rocprofv3 with OpenMP*

## 2.9 Output Management

### 2.9.1 File Organization

```
# Specify output directory
rocprofv3 --runtime-trace --output-directory ./results --output-file my_trace -- ./
↪ your_app

# Generate configuration file
rocprofv3 --runtime-trace --output-config -- ./your_app
```

**Documentation:** *Using rocprofv3* (I/O options section)

## 2.10 Common Use Cases

### 2.10.1 Basic Performance Analysis

```
# Quick performance overview
rocprofv3 --runtime-trace --summary -- ./your_app
```

**Use case:** Get a high-level view of application performance

### 2.10.2 Detailed Kernel Analysis

```
# Detailed kernel profiling with counters
rocprofv3 --kernel-trace --pmc SQ_WAVES,SQ_INSTS_VALU,TCP_PERF_SEL_TOTAL_CACHE_ACCESSES -
↪ - ./your_app
```

**Use case:** Analyze specific kernel performance bottlenecks

### 2.10.3 Memory Transfer Analysis

```
# Focus on memory operations
rocprofv3 --memory-copy-trace --memory-allocation-trace -- ./your_app
```

**Use case:** Optimize data movement between CPU and GPU

### 2.10.4 Timeline Visualization

```
# Generate timeline for visualization tools
rocprofv3 --runtime-trace -- ./your_app

# Convert to Perfetto format
rocpd2pftrace -i hostname/pid_results.db -o perfetto_trace
```

**Use case:** Visualize execution timeline in Perfetto or similar tools

## 2.11 Installation and Setup

**Installation Documentation:** [Installing ROCprofiler-SDK](#)

**API Reference:** [Tool library](#)

**Samples and Examples:** [Samples](#)

## 2.12 Troubleshooting Quick Tips

1. **Permission Issues:** Ensure proper access to GPU devices and `/dev/kfd`
2. **Counter Collection Fails:** Use `rocprofv3-avail pmc-check` to verify counter compatibility
3. **Large Output Files:** Use `--minimum-output-data` to set file size thresholds
4. **Signal Handling:** Use `--disable-signal-handlers` if conflicts with application handlers
5. **ROCm Path Issues:** Use `--rocm-root` to specify custom ROCm installation paths

For comprehensive documentation on each feature, refer to the detailed sections linked throughout this guide.



## ROCPROFILER-SDK SAMPLES

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

### 3.1 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
```

### 3.2 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
```

### 3.3 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.



## 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>
```

### 4.1 Command-line options

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

Table 1: rocprofv3 options

Purpose	Option	Description
I/O options	<p>-i INPUT   --input INPUT</p> <p>-o OUTPUT_FILE  </p> <p>--output-file OUTPUT_FILE</p> <p>-d OUTPUT_DIRECTORY  </p> <p>--output-directory OUTPUT_DIRECTORY</p> <p>-f {csv,json,pftrace,otf2,rocpd} [ {csv,json,pftrace,otf2,rocpd} ... ]   --output-format {csv,json,pftrace,otf2,rocpd} [ {csv,json,pftrace,otf2,rocpd} ... ]</p> <p>--output-config [BOOL]</p> <p>--log-level {fatal,error,warning,info,trace,env}</p> <p>-E EXTRA_COUNTERS  </p> <p>--extra-counters EXTRA_COUNTERS</p>	<p>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.</p> <p>Specifies output file name. If nothing is specified, the default path is %hostname%/%pid%.</p> <p>Specifies the output path for saving the output files. If nothing is specified, the default path is %hostname%/%pid%.</p> <p>Specifies output format. Supported formats: CSV, JSON, PFTrace, OTF2 and rocpd.</p> <p>Generates a configuration output file containing the resolved rocprofv3 settings and options used for the profiling session.</p> <p>Sets the desired log level.</p> <p>Specifies the path to a YAML file consisting of extra counter definitions.</p>
Process attachment	<p>-p PID   --pid PID   --attach PID</p>	<p>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.</p>
Aggregate tracing	<p>-r [BOOL]   --runtime-trace [BOOL]</p> <p>-s [BOOL]   --sys-trace [BOOL]</p>	<p>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.</p> <p>Collects tracing data for HIP API, HSA API, marker (ROCTx) API, RCCL API, memory operations (copies, scratch, and allocations), and kernel dispatches.</p>

To see exhaustive list of `rocprofv3` options:

```
rocprofv3 -h
rocprofv3 --help
```

To display version information for `rocprofv3`:

```
rocprofv3 -v
rocprofv3 --version
```

The version command provides comprehensive build and system information including:

```
$ rocprofv3 -v
      version: 1.0.0
  git_revision: a1b2c3d4e5f6789012345678901234567890abcd
  library_arch: x86_64-linux-gnu
    system_name: Linux
  system_processor: x86_64
    system_version: 6.8.0-57-generic
      compiler_id: GNU
  compiler_version: 11.4.0
      rocm_version: 6.2.0
```

## 4.2 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](#).

### 4.2.1 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 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_RUNI	hipGetDeviceCount	15	15	6	1055015440617618	1055015539800733
HIP_RUNI	hipSetDevice	15	15	7	1055015539819503	1055015539821693
HIP_RUNI	hipDeviceSynchronize	15	15	8	1055015539832333	1055015539840903
HIP_RUNI	hipStreamCreateWithFlags	15	15	9	1055015539861673	1055015865247140
HIP_RUNI	hipHostMalloc	15	15	10	1055015865309761	1055015865849494
HIP_RUNI	hipHostMalloc	15	15	11	1055015865850944	1055015866265546
HIP_RUNI	hipHostMalloc	15	15	12	1055015866266646	1055015867082900
HIP_RUNI	hipMallocAsync	15	15	13	1055015867356542	1055015867662314
HIP_RUNI	hipMallocAsync	15	15	14	1055015867664174	1055015867937465
HIP_RUNI	hipMallocAsync	15	15	15	1055015867938815	1055015868219987
HIP_RUNI	hipMemcpyAsync	15	15	16	1055015868240137	1055015917307652
HIP_RUNI	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: 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 4: HIP runtime api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_RUNT	hipGet-Device-PropertiesR0600	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*.

## 4.2.2 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 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_1	197	197	3	1507843974754471	1507843974755014
HSA_AME	hsa_amd_r	197	197	5	1507843974761705	1507843974762398
HSA_AME	hsa_amd_r	197	197	6	1507843974763901	1507843974764030
HSA_AME	hsa_amd_r	197	197	7	1507843974765121	1507843974765224
HSA_AME	hsa_amd_r	197	197	8	1507843974766196	1507843974766328
HSA_AME	hsa_amd_r	197	197	9	1507843974767534	1507843974767641
HSA_AME	hsa_amd_r	197	197	10	1507843974768639	1507843974768779
HSA_AME	hsa_amd_a	197	197	4	1507843974758768	1507843974769238
HSA_COR	hsa_agent_1	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 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_	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_c	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*.

### 4.2.3 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*.

### 4.2.4 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 in-built 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
```

### 4.2.5 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 7: Kernel trace

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

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

### 4.2.6 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 8: Memory copy trace

Kind	Direction	Stream_Id	Source_Agent	Destination_Agent	Correlation_Id	Start_Timestamp	End_Timestamp
MEM-ORY_COI	MEM-ORY_COI	0	Agent 0	Agent 4	1	1057963336487172	1057963336564212
MEM-ORY_COI	MEM-ORY_COI	0	Agent 0	Agent 4	2	1057963336783973	1057963336859334
MEM-ORY_COI	MEM-ORY_COI	0	Agent 4	Agent 0	23	1057963497396292	1057963497471732
MEM-ORY_COI	MEM-ORY_COI	0	Agent 4	Agent 0	24	1057963498099125	1057963498200446

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

### 4.2.7 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 9: Memory allocation trace

Kind	Operation	Agent_Id	Allocation_Size	Address	Correlation_Id	Start_Timestamp	End_Timestamp
MEM-ORY_ALI	MEM-ORY_ALI	Agent 0	1024	0x00007ff	1	816098791282238	816098791339655
MEM-ORY_ALI	MEM-ORY_ALI	Agent 0	1024	0x00007ff	2	816098791350331	816098791386746
MEM-ORY_ALI	MEM-ORY_ALI		0	0x00007ff	7	816098791533678	816098791678768
MEM-ORY_ALI	MEM-ORY_ALI		0	0x00007ff	8	816098791681482	816098791873422

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

### 4.2.8 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.

### 4.2.9 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.

### 4.2.10 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 10: Scratch memory trace

Kind	Operation	Agent	Queue	Thread	Alloc_Fla	Start_Timestamp	End_Timestamp	Allocation_Size
SCRAT	SCRAT	Agent 1	1	5007	1	93225608906863	93225608938858	173015040
SCRAT	SCRAT	Agent 1	1	5007	1	93225609034561	93225609034878	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609041150	93225609062327	1131626496
SCRAT	SCRAT	Agent 1	1	5007	1	93225609352143	93225609353099	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609359557	93225609360209	45056000
SCRAT	SCRAT	Agent 1	1	5007	1	93225609390269	93225609390683	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609396239	93225609396668	67764224
SCRAT	SCRAT	Agent 1	1	5007	1	93225609429533	93225609429925	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609440437	93225609440871	45056000
SCRAT	SCRAT	Agent 1	1	5007	1	93225609468592	93225609468988	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609479396	93225609479836	1131626496
SCRAT	SCRAT	Agent 1	1	5007	1	93225609741667	93225609742611	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225609747829	93225609748464	1025024000
SCRAT	SCRAT	Agent 1	1	5007	1	93225610006694	93225610007631	0
SCRAT	SCRAT	Agent 1	1	5007	1	93225610012520	93225610013182	1025024000
SCRAT	SCRAT	Agent 1	1	5007	1	93225610274420	93225610275359	0

For the description of the fields in the output file, see [Output file fields](#).

#### 4.2.11 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 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	ncclGroupEnc	1834151	1834151	1490	18414166823575	18414380520973
RCCL_API	ncclCommInitAll	1834151	1834151	1477	18414166402665	18414380522536
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89098	18414380660695	18414380661652
RCCL_API	ncclAllReduce	1834151	1834151	89097	18414380653860	18414380693574
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89108	18414380694631	18414380694659
RCCL_API	ncclAllReduce	1834151	1834151	89107	18414380694212	18414380704722
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89117	18414380706650	18414380706677
RCCL_API	ncclAllReduce	1834151	1834151	89116	18414380705574	18414380715055
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89126	18414380715749	18414380715774
RCCL_API	ncclAllReduce	1834151	1834151	89125	18414380715463	18414380723944
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89135	18414380724688	18414380724715
RCCL_API	ncclAllReduce	1834151	1834151	89134	18414380724395	18414380732209
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89154	18414380746383	18414380746411
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89157	18414380749863	18414380749889
RCCL_API	ncclCommGetAsynchError	1834151	1834151	89160	18414380751671	18414380751696

#### 4.2. Application tracing

29

RCCL_API	ncclCommGetAsynchError	1834151	1834151	89163	18414380753326	18414380753353
----------	------------------------	---------	---------	-------	----------------	----------------

## 4.2.12 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 12: rocDecode trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
ROCDE-CODE_API	rocDec-Create-VideoParser	41688	41688	583	615449881677279	615449882001583
ROCDE-CODE_API	rocDecGet-Decoder-Caps	41688	41688	584	615449882016054	615449882163756
ROCDE-CODE_API	rocDecGet-Decoder-Caps	41688	41688	588	615449886038750	615449886050880
ROCDE-CODE_API	rocDec-CreateDecoder	41688	41688	591	615449886084210	615450756910310
ROCDE-CODE_API	rocDecDecode-Frame	41688	41688	595	615450757036042	615450767147413
ROCDE-CODE_API	rocDecGet-DecodeStatus	41688	41688	812	615450836779385	615450836779575

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

## 4.2.13 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 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_	rocJpegGetImage-Info	41884	41884	504	1286306248684203	1286306248686556

#### 4.2.14 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.

Read in detail about process attachment in `using-rocprofv3-process-attachment`.

#### 4.2.15 Post-processing tracing options

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

##### 4.2.15.1 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 14: HIP stats

Name	Calls	TotalDurationNs	AverageNs	Per-centage	MinNs	MaxNs	StdDev
hip-Stream-Create-With-Flags	4	262497406	65624351.500000	85.15	3991286	24912184	122332531.343496
hipGet-Device-Count	1	32505687	32505687.000000	10.54	32505687	32505687	0.00000000e+00
hipHost-Malloc	12	6096409	508034.083333	1.98	443793	548024	39236.753678
hipFree	12	1994421	166201.750000	0.6470	7790	1036046	299086.860470
hip-Mem-cpyAsync	12	1368378	114031.500000	0.4439	2490	764044	249308.051619
hipMal-locAsync	12	927255	77271.250000	0.3008	51540	107671	20487.475966
hip-Stream-Syn-chronize	12	870486	72540.500000	0.2824	140	866606	250065.900069
hipLaunch-ernel	16	692734	43295.875000	0.2247	1000	670044	167133.656647
hip-StreamDe-stroy	4	619905	154976.250000	0.2011	92901	339252	122852.320356
hipDe-viceSyn-chronize	4	404252	101063.000000	0.1311	570	385212	189518.505401
hipHost-Free	12	271202	22600.166667	0.0880	11950	34950	7480.268600
__hipReg-isterFat-Binary	1	9000	9000.000000	2.920e-03	9000	9000	0.00000000e+00
__hipReg-ister-Func-tion	4	6150	1537.500000	1.995e-03	230	5370	2555.091323
__hip-Push-Call-Config-uration	16	2460	153.750000	7.980e-04	70	1140	267.503894
__hip-Pop-Call-Config-uration	16	2000	125.000000	6.488e-04	70	680	151.613544
hipGet-LastEr-ror	16	1270	79.375000	4.120e-04	50	440	96.295985
hipSet-Device	1	660	660.000000	2.141e-04	660	660	0.00000000e+00

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

Table 15: Domain stats

Name	Calls	TotalDurationNs	AverageNs	Per-centage	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*.

#### 4.2.15.2 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	212088348	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	0.066095	83953	132851	3.458e+04
hipMalloc	HIP_API	2	51981	2.599e+04	0.015991	10502	41479	2.190e+04

#### 4.2.15.3 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.

#### 4.2.15.4 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	0.060929	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	43099	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

#### 4.2.15.5 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	3.06e+04
hsa_signal_wait_scacquire	HSA_API	26	853122	3.281e+04	0.229587	2530	100782	3.94e+04
hsa_executable_load_agent_code_object	HSA_API	2	616175	3.081e+05	0.165821	254476	361699	5.82e+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								107		↪
↪160889		1.504e+03		0.043297		80		5650		1.
↪475e+03										
hsa_code_object_reader_create_from_memory								2		↪
↪151314		7.566e+04		0.040721		32121		119193		6.
↪157e+04										
hsa_signal_load_relaxed								1296		↪
↪137626		1.062e+02		0.037037		20		2930		2.
↪712e+02										
hsa_signal_destroy								618		↪
↪111224		1.800e+02		0.029932		40		1540		2.
↪429e+02										
hsa_agent_get_info								65		↪
↪77472		1.192e+03		0.020849		30		47121		6.
↪341e+03										
hsa_amd_signal_create								512		↪
↪61290		1.197e+02		0.016494		40		930		1.
↪559e+02										
hsa_amd_signal_async_handler								24		↪
↪52641		2.193e+03		0.014166		1180		4020		9.
↪252e+02										
hsa_executable_iterate_symbols								14		↪
↪52521		3.752e+03		0.014134		2740		6940		1.
↪105e+03										
hsa_amd_memory_copy_engine_status								18		↪
↪47370		2.632e+03		0.012748		260		7990		2.
↪274e+03										
hsa_iterate_agents								1		↪
↪41391		4.139e+04		0.011139		41391		41391		0.
↪000e+00										
hsa_executable_create_alt								2		↪
↪40470		2.024e+04		0.010891		7530		32940		1.
↪797e+04										
hsa_isa_get_info_alt								2		↪
↪30391		1.520e+04		0.008179		2490		27901		1.
↪797e+04										
hsa_signal_silent_store_relaxed								48		↪
↪24920		5.192e+02		0.006706		20		4570		7.
↪120e+02										
hsa_amd_agent_iterate_memory_pools								5		↪
↪20221		4.044e+03		0.005442		2561		8600		2.
↪574e+03										
hsa_queue_add_write_index_screlease								56		↪
↪7270		1.298e+02		0.001956		30		2310		3.
↪471e+02										
hsa_amd_profiling_set_profiler_enabled								4		↪
↪5600		1.400e+03		0.001507		1370		1470		4.
↪690e+01										
hsa_executable_symbol_get_info								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	
↪455e+05									1.
hipDeviceSynchronize				HIP_API		4		↪	
↪50663		1.267e+04		0.008887		510		23621	
↪554e+03									9.
__hipRegisterFatBinary				HIP_API		1		↪	
↪43811		4.381e+04		0.007685		43811		43811	
↪000e+00									0.
__hipPushCallConfiguration				HIP_API		32		↪	
↪6250		1.953e+02		0.001096		60		3640	
↪308e+02									6.
__hipPopCallConfiguration				HIP_API		32		↪	
↪4780		1.494e+02		0.000838		60		2520	
↪340e+02									4.
hipGetLastError				HIP_API		32		↪	
↪4471		1.397e+02		0.000784		60		2381	
↪092e+02									4.
hipSetDevice				HIP_API		1		↪	
↪2570		2.570e+03		0.000451		2570		2570	
↪000e+00									0.

ROCPROFV3 KERNEL\_DISPATCH SUMMARY:

	CALLS	DURATION (nsec)	AVERAGE (nsec)	PERCENT (INC)	DOMAIN
↪(nsec)	MAX (nsec)	STDDEV			MIN
-----					
↪	↪	↪	↪	↪	↪
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
↪ 207e+03				↪ 8.

ROCPROFV3 MEMORY\_ALLOCATION SUMMARY:

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

ROCPROFV3 SUMMARY:

	NAME	DOMAIN	CALLS	DURATION (nsec)	AVERAGE (nsec)	PERCENT (INC)
MIN (nsec)	STDDEV		MAX (nsec)			
	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
↳	76707894		76707894		0.000e+00	
hipMemcpyAsync					HIP_API	↳
↳		24		56109444		2.338e+06
↳	11640		55299811		1.128e+07	
hsa_amd_memory_async_copy_on_engine					HSA_API	↳
↳		24		55617052		2.317e+06
↳	7580		55195188		1.126e+07	
hsa_amd_memory_pool_allocate					HSA_API	↳
↳		67		26428438		3.945e+05
↳	1510		857592		1.782e+05	
MEMORY_ALLOCATION_ALLOCATE					MEMORY_	↳
↳	ALLOCATION		67		26314096	
↳	950		856812		1.785e+05	
hipHostMalloc					HIP_API	↳
↳		24		13007523		5.420e+05
↳	416631		866382		1.206e+05	
hipMallocAsync					HIP_API	↳
↳		24		7304847		3.044e+05
↳	275397		353719		2.207e+04	
hsa_amd_memory_pool_free					HSA_API	↳
↳		72		5176173		7.189e+04
↳	290		170374		3.903e+04	
MEMORY_ALLOCATION_FREE					MEMORY_	↳
↳	ALLOCATION		72		5138913	
↳	20		166234		3.882e+04	
MEMORY_COPY_HOST_TO_DEVICE					MEMORY_	↳
↳	COPY		16		3691929	
↳	74842		284487		6.265e+04	
hipHostFree					HIP_API	↳
↳		24		2786484		1.161e+05
↳	72242		221646		4.606e+04	
hipStreamDestroy					HIP_API	↳
↳		8		2137924		2.672e+05
↳	221596		377469		5.489e+04	
hipLaunchKernel					HIP_API	↳
↳		32		2080214		6.501e+04
↳	8850		1608721		2.819e+05	
hipFree					HIP_API	↳
↳		24		1572948		6.554e+04
↳	2130		186994		4.815e+04	
hipStreamSynchronize					HIP_API	↳
↳		24		1452706		6.053e+04
↳	20810		135803		3.469e+04	
hsa_executable_freeze					HSA_API	↳
↳		2		964125		4.821e+05
↳	437471		526654		6.306e+04	
hsa_signal_wait_scacquire					HSA_API	↳
↳		26		853122		3.281e+04
↳	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					↳

## 4.2.16 Configuration Output

The `--output-config` option generates a comprehensive configuration output file that contains all resolved `rocprofv3` settings and options used during a profiling session. This feature is essential for debugging, reproducibility, and configuration validation.

To generate a configuration output file during profiling:

```
rocprofv3 --output-config --hip-trace -- <application_path>
```

This command generates a configuration file (typically `<process_id>_config.json`) alongside the regular profiling output files.

The generated JSON configuration file contains detailed information about the profiling session and is structured with a `rocprofiler-sdk-tool` array containing comprehensive metadata and configuration details.

The metadata section includes essential session information such as process ID (`pid`), initialization and finalization timestamps (`init_time`, `fini_time`), the exact command executed, and detailed build specifications. The build specification contains version information, compiler details, git revision, system architecture, and kernel version, providing complete context for reproducing the environment.

The config section is the most comprehensive part, containing all profiling options with their resolved boolean and numerical values. This includes tracing options like `hip_runtime_api_trace`, `hip_compiler_api_trace`, `kernel_trace`, `hsa_core_api_trace`, `memory_copy_trace`, and many others. It also shows advanced configuration like PC sampling settings (`pc_sampling_method`, `pc_sampling_interval`), filtering options (`kernel_filter_include`, `kernel_filter_exclude`), output formatting choices (`csv_output`, `json_output`, `pftrace_output`), and performance tuning parameters.

The environment section captures all environment variables active during the profiling session, including system variables such as `SHELL`, `COLORTERM`, `HOSTNAME`, and ROCm-specific variables, providing complete environmental context for reproduction.

Example configuration output structure:

```
{
  "rocprofiler-sdk-tool": [
    {
      "metadata": {
```

(continues on next page)

(continued from previous page)

```

"pid": 213524,
"init_time": 682678344984459,
"fini_time": 682678842290172,
"config": {
  "hip_runtime_api_trace": true,
  "hip_compiler_api_trace": true,
  "kernel_trace": false,
  "hsa_core_api_trace": false,
  "memory_copy_trace": false,
  "counter_collection": false,
  "kernel_filter_include": ".*",
  "demangle": true,
  "minimum_output_bytes": 0,
  "csv_output": true,
  "json_output": false,
  "output_path": "out",
  "output_file": "1a2b3c4d5e6f/213524"
},
"command": ["/MatrixTranspose"],
"build_spec": {
  "version_major": 1,
  "version_minor": 0,
  "compiler_id": "GNU",
  "compiler_version": "11.4.0",
  "git_revision": "a1b2c3d4e5f6789012345678901234567890abcd",
  "system_name": "Linux",
  "system_processor": "x86_64"
},
"environment": {
  "SHELL": "/bin/bash",
  "COLORTERM": "truecolor",
  "HOSTNAME": "1a2b3c4d5e6f",
  "ROCM_ROOT": "/opt/rocm-6.4.2",
  "ROCM_VERSION": "6.4.2",
  "BUILD_NUM": "12345",
  "ROCPROF_OUTPUT_PATH": "out",
  "ROCPROF_OUTPUT_CONFIG_FILE": "1",
  "ROCPROF_OUTPUT_FORMAT": "csv",
  "ROCPROF_HIP_COMPILER_API_TRACE": "1",
  "ROCPROF_HIP_RUNTIME_API_TRACE": "1",
  ".... Output truncated for brevity ...."
}
}
}
]
}

```

The configuration output file provides complete transparency into rocprofv3 operation, documenting all settings, defaults, and environmental context required for profiling sessions.

## 4.2.17 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:

```
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:

```
{
  "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"
    }
  ]
}
```

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.

#### 4.2.18 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.

## 4.3 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
```

Sample output for the list-avail command:

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](#).

### 4.3.1 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"]
    },
    {
      "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"
```

(continues on next page)

(continued from previous page)

```
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>
```

### 4.3.2 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.

### 4.3.3 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
```

(continues on next page)

(continued from previous page)

```

    - 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.

#### 4.3.4 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 16: Counter collection

Cor- re- la- tion_	Dis- patc	Age	Que	Pro- cess	Thre	Grid	Ker- nel_	Ker- nel_	Wor grou	LDS	Scr	VGF	Ac- cum	SGF	Cou	Cou	Star	Ende	Start	stamp
1	1	Age 1	1	156C	156C	1048	17	void_	64	0	0	8	0	16	SQ_	1638	320C	32000	0988084	183232
								ad- di- tion_ float cons float cons int, int)												
2	2	Age 1	1	156C	156C	1048	20	sub- tract	64	0	0	8	0	16	SQ_	1638	320C	32000	0988086	138794
								float cons float cons int, int)												
3	3	Age 1	1	156C	156C	1048	19	mul- ti- ply_]	64	0	0	8	0	16	SQ_	1638	320C	32000	0988087	93025
								float cons float cons int, int)												
4	4	Age 1	1	156C	156C	1048	18	di- vide_	64	0	0	12	0	16	SQ_	1638	320C	32000	0988089	27550
								float cons float cons int, int)												

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

### 4.3.5 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
    }
  ]
}
```

## 4.4 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.

### 4.4.1 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 pfttrace` 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 pfttrace -- <application_path>
```

The generated Perfetto trace file can be opened in the [Perfetto UI](#).

**Figure 1:** Generic perfetto visualization

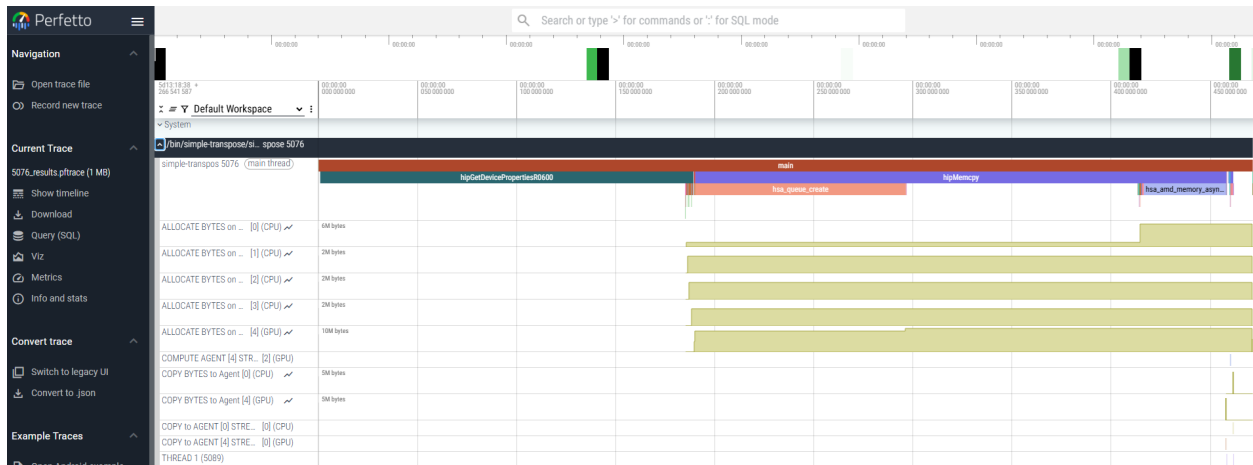
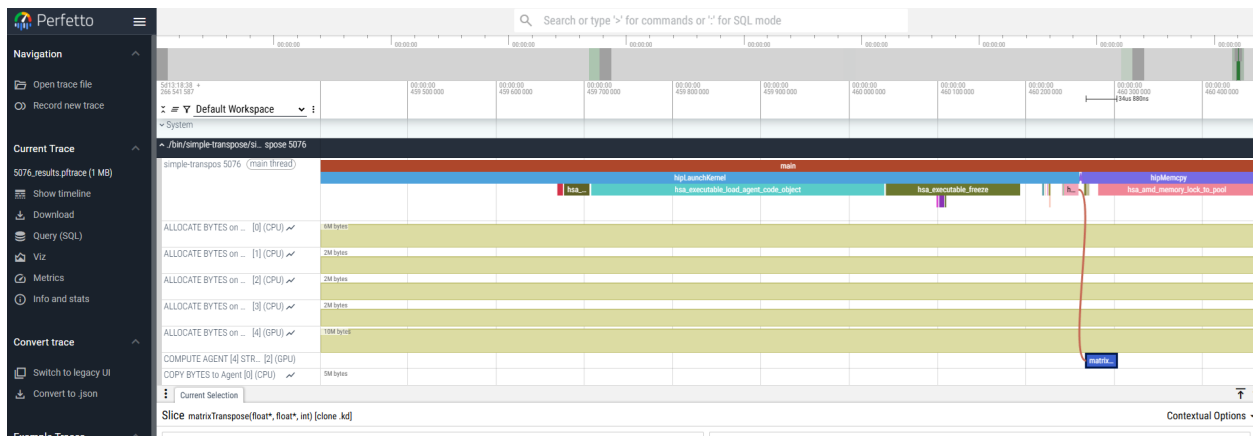


Figure 2: Visualization of ROCm flow data in Perfetto



#### 4.4.2 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.

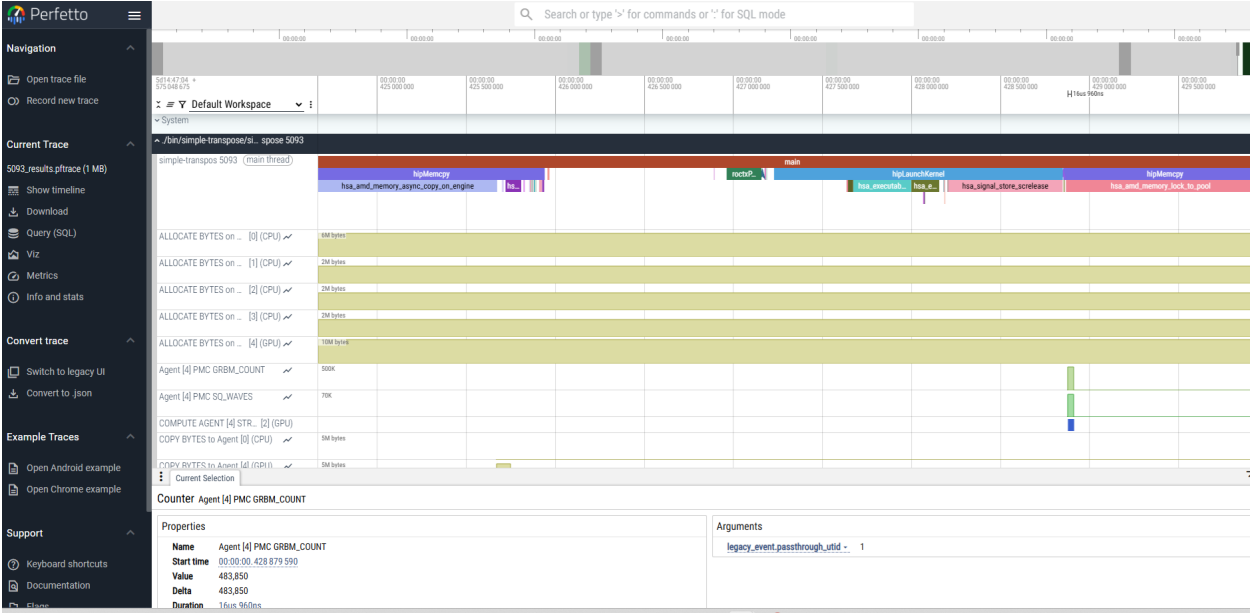
To generate a Perfetto trace file with counter data, use:

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



### 4.4.3 Scratch Memory Visualization in Perfetto

When using the `--scratch-memory-trace` option with Perfetto output format, ROCProfiler SDK creates visualization tracks for scratch memory usage. Scratch memory operations are displayed as counter tracks organized by agent (GPU), allowing you to monitor the scratch memory allocation patterns during kernel execution.

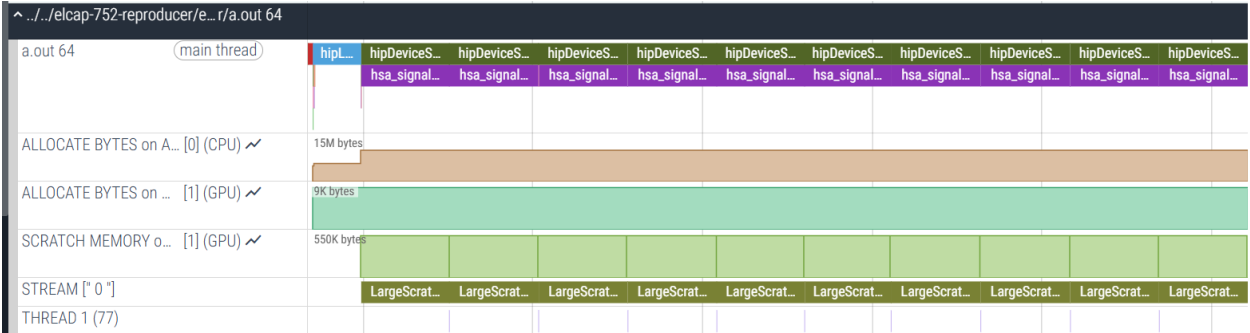
To generate a Perfetto trace file that includes scratch memory visualization:

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

In the Perfetto UI, scratch memory appears as counter tracks that show:

- **Allocation peaks:** Each peak represents scratch memory allocation for a kernel execution
- **Memory usage over time:** The height of each peak indicates the amount of memory allocated (typically in KB)
- **Allocation/deallocation pattern:** You can observe when memory is allocated at kernel start and freed at kernel end

For applications with multiple kernel iterations, you'll see multiple peaks in the scratch memory track, with each peak corresponding to a kernel execution. This visualization helps identify scratch memory usage patterns and potential optimization opportunities.



For comprehensive GPU execution insights, combine scratch memory tracing with kernel tracing:



To specify the minimum output data threshold, use the `--minimum-output-data` option followed by the size in KB:

```
rocprofv3 --minimum-output-data 100 --hip-trace --output-format csv -- <application_path>
```

The preceding command only generates output files if the HIP trace data is larger than 100 KB.

#### Example scenarios:

##### Scenario 1: Filtering out applications with minimal GPU activity

```
# Only generate output if kernel trace data > 50 KB
rocprofv3 --minimum-output-data 50 --kernel-trace --output-format csv -- <application_
↳path>
```

##### Scenario 2: Batch profiling with meaningful data collection

```
# For system tracing, only output files if data > 1 MB
rocprofv3 --minimum-output-data 1024 --sys-trace --output-format pfttrace -- <application_
↳path>
```

#### Using with input files:

You can also specify this option in YAML or JSON input files:

```
jobs:
- hip_trace: true
  kernel_trace: true
  minimum_output_data: 100
  output_format: ["csv", "json"]
  output_directory: "filtered_results"
```

```
{
  "jobs": [
    {
      "hip_trace": true,
      "kernel_trace": true,
      "minimum_output_data": 100,
      "output_format": ["csv", "json"],
      "output_directory": "filtered_results"
    }
  ]
}
```

#### Important notes:

- The threshold applies to the raw profiling data size, not the final output file size
- If multiple output formats are specified, the threshold check applies to each format independently
- A value of 0 (default) means all output files are generated regardless of size
- This option works with all tracing and counter collection modes

This feature is especially valuable in automated testing environments where you want to ensure that only applications with meaningful GPU activity generate profiling outputs, reducing storage overhead and simplifying result analysis.

## 4.6.2 Signal handler control

The `--disable-signal-handlers` option provides control over signal handling behavior in `rocprofv3`, allowing you to manage how the profiler responds to system signals like `SIGSEGV`, `SIGTERM`, and others.

By default, `rocprofv3` installs its own signal handlers to ensure proper cleanup and data collection when the application encounters errors or is terminated. However, in some scenarios, you may want the application's own signal handlers to take precedence.

When `--disable-signal-handlers` is set to `true`, `rocprofv3` disables the prioritization of its signal handlers over application-installed signal handlers. This means:

- If your application has custom signal handlers for `SIGSEGV`, `SIGTERM`, or similar signals, those handlers will be executed instead of `rocprofv3`'s handlers
- The application maintains full control over signal handling behavior
- `rocprofv3` will still attempt to collect and save profiling data when possible

**Important note:** Even with this option enabled, the underlying `glog` library may still install signal handlers that provide stack backtraces for debugging purposes.

### Basic usage:

```
rocprofv3 --disable-signal-handlers --hip-trace --output-format csv -- <application_path>
```

The preceding command disables `rocprofv3` signal handler prioritization, allowing the application's signal handlers to take precedence.

### Example scenarios:

#### Scenario 1: Application with custom crash handling

```
# For applications that implement custom crash reporting or recovery
rocprofv3 --disable-signal-handlers --sys-trace --output-format pfttrace -- ./my_app_with_
↳ custom_handlers
```

#### Scenario 2: Debugging applications with existing signal handlers

```
# When debugging applications that rely on specific signal handling behavior
rocprofv3 --disable-signal-handlers --kernel-trace --pmc SQ_WAVES -- ./debug_application
```

#### Scenario 3: Integration with testing frameworks

```
# For test frameworks that need to handle signals for test orchestration
rocprofv3 --disable-signal-handlers --runtime-trace --output-directory test_results -- ./
↳ test_suite
```

### Using with input files:

You can also specify this option in YAML or JSON input files:

```
jobs:
- hip_trace: true
  kernel_trace: true
  disable_signal_handlers: true
  output_format: ["csv", "json"]
  output_directory: "custom_signal_handling"
```

```
{
  "jobs": [
    {
      "hip_trace": true,
      "kernel_trace": true,
      "disable_signal_handlers": true,
      "output_format": ["csv", "json"],
      "output_directory": "custom_signal_handling"
    }
  ]
}
```

**When to use this option:****Use when:**

- Your application has custom signal handlers that must execute
- You're integrating with testing frameworks that manage signals
- Debugging applications where signal handling behavior is critical
- Working with applications that implement custom crash reporting

**Avoid when:**

- You want rocprofv3 to provide maximum protection against data loss
- Your application doesn't have custom signal handlers
- You're doing standard profiling where signal handling isn't a concern

**Example: Application with custom SIGSEGV handler**

If your application has a custom segmentation fault handler:

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

void custom_sigsegv_handler(int sig) {
    printf("Custom SIGSEGV handler called\n");
    // Custom crash reporting logic
    exit(1);
}

int main() {
    signal(SIGSEGV, custom_sigsegv_handler);

    // Application code that might trigger SIGSEGV
    return 0;
}
```

Use `--disable-signal-handlers` to ensure your custom handler executes:

```
rocprofv3 --disable-signal-handlers --hip-trace -- ./app_with_custom_handler
```

**Troubleshooting:**

- If profiling data appears incomplete with this option enabled, check if your application's signal handlers are properly saving or flushing data

- Consider implementing explicit rocprofv3 cleanup calls in your application's signal handlers if data integrity is important
- Monitor application behavior to ensure custom signal handling doesn't interfere with profiling data collection

This option provides the flexibility needed for complex applications and testing environments while maintaining rocprofv3's core profiling functionality.

### 4.6.3 Library preloading

The `--preload` option allows you to specify additional libraries to prepend to the `LD_PRELOAD` environment variable. This is particularly useful when working with sanitizer libraries, debugging tools, or other instrumentation libraries that need to be loaded before the application starts.

`LD_PRELOAD` is a powerful mechanism in Linux that allows you to load shared libraries before any other libraries, effectively intercepting and overriding function calls. The `--preload` option in rocprofv3 provides a convenient way to manage this without manually setting environment variables.

#### Basic usage:

```
rocprofv3 --preload /path/to/library.so --hip-trace --output-format csv -- <application_
↳path>
```

The preceding command preloads the specified library and enables HIP tracing.

#### Example scenarios:

##### Scenario 1: Using AddressSanitizer (ASan)

```
# Preload AddressSanitizer for memory error detection
rocprofv3 --preload /usr/lib/x86_64-linux-gnu/libasan.so.5 --sys-trace -- ./my_
↳application
```

##### Scenario 2: Using ThreadSanitizer (TSan)

```
# Preload ThreadSanitizer for race condition detection
rocprofv3 --preload /usr/lib/x86_64-linux-gnu/libtsan.so.0 --kernel-trace --pmc SQ_WAVES_
↳-- ./threaded_app
```

##### Scenario 3: Multiple preloaded libraries

```
# Preload multiple libraries (custom profiler and sanitizer)
rocprofv3 --preload /opt/custom/libprofiler.so /usr/lib/libasan.so --runtime-trace -- ./
↳complex_app
```

##### Scenario 4: Using MemorySanitizer (MSan)

```
# Preload MemorySanitizer for uninitialized memory detection
rocprofv3 --preload /usr/lib/x86_64-linux-gnu/libmsan.so.0 --hip-trace -- ./memory_
↳intensive_app
```

#### Using with input files:

You can also specify this option in YAML or JSON input files:

```
jobs:
  - hip_trace: true
    kernel_trace: true
```

(continues on next page)

(continued from previous page)

```

preload:
  - "/usr/lib/x86_64-linux-gnu/libasan.so.5"
  - "/opt/custom/libprofiler.so"
output_format: ["csv"]

```

```

{
  "jobs": [
    {
      "hip_trace": true,
      "kernel_trace": true,
      "preload": [
        "/usr/lib/x86_64-linux-gnu/libasan.so.5",
        "/opt/custom/libprofiler.so"
      ],
      "output_format": ["csv"]
    }
  ]
}

```

**Common use cases:**

**Sanitizer libraries:** - AddressSanitizer (`libasan.so`) for memory error detection - ThreadSanitizer (`libtsan.so`) for race condition detection - MemorySanitizer (`libmsan.so`) for uninitialized memory detection - UndefinedBehaviorSanitizer (`libubsan.so`) for undefined behavior detection

**Debugging and profiling tools:**

- Custom memory allocators (`jemalloc`, `tcmalloc`)
- Performance profiling libraries
- Custom instrumentation libraries
- Mock libraries for testing

**Third-party analysis tools:**

- Valgrind replacement libraries
- Custom logging frameworks
- Security analysis tools

**Library order considerations:**

The order of libraries in `--preload` matters as they are processed in the order specified:

```

# Library1 will be loaded before Library2
rocprofv3 --preload /path/to/library1.so /path/to/library2.so --hip-trace -- ./app

```

**Environment variable interaction:**

The `--preload` option works alongside existing `LD_PRELOAD` settings:

```

# If LD_PRELOAD is already set, --preload libraries are prepended
export LD_PRELOAD="/existing/library.so"
rocprofv3 --preload /new/library.so --hip-trace -- ./app
# Effective LD_PRELOAD: "/new/library.so:/existing/library.so"

```

**Troubleshooting:**

- **Library not found:** Ensure the library path is correct and the library exists
- **Symbol conflicts:** Check for conflicting symbols between preloaded libraries
- **Performance impact:** Sanitizers can significantly slow down execution
- **Memory usage:** Some tools like AddressSanitizer increase memory consumption substantially

### 4.6.4 ROCm root path configuration

The `--rocm-root` option allows you to specify a custom ROCm installation directory instead of using the default relative path detection. This is useful when working with multiple ROCm installations, custom builds, or non-standard installation locations.

By default, `rocprofv3` automatically detects the ROCm installation path relative to its own location. However, in some environments, you may need to explicitly specify which ROCm installation to use.

**Basic usage:**

```
rocprofv3 --rocm-root /opt/custom-rocm --hip-trace --output-format csv -- <application_
↳path>
```

The preceding command uses the ROCm installation located at `/opt/custom-rocm`.

**Example scenarios:**

**Scenario 1: Multiple ROCm versions**

```
# Use ROCm 5.7.0 specifically
rocprofv3 --rocm-root /opt/rocm-5.7.0 --sys-trace -- ./app_for_rocm_5_7

# Use ROCm 6.0.0 for comparison
rocprofv3 --rocm-root /opt/rocm-6.0.0 --sys-trace -- ./app_for_rocm_6_0
```

**Scenario 2: Custom ROCm build**

```
# Use custom ROCm build with debugging symbols
rocprofv3 --rocm-root /home/developer/rocm-debug-build --kernel-trace --pmc SQ_WAVES -- .
↳/debug_app
```

**Scenario 3: Development environment**

```
# Use locally built ROCm for development
rocprofv3 --rocm-root /workspace/rocm-dev --runtime-trace -- ./test_application
```

**Scenario 4: Container environments**

```
# Use ROCm mounted at custom location in container
rocprofv3 --rocm-root /usr/local/rocm --hip-trace -- ./containerized_app
```

**Directory structure requirements:**

The specified ROCm root path should contain the standard ROCm directory structure:

```
/opt/custom-rocm/
├─ bin/           # ROCm executables
├─ lib/          # ROCm libraries
```

(continues on next page)

(continued from previous page)

```
├─ include/      # ROCm headers
├─ share/       # Shared resources
└─ ...
```

**Using with input files:**

This option is typically used from the command line, but can be specified in wrapper scripts:

```
#!/bin/bash
# profile_with_custom_rocm.sh
ROCM_PATH="/opt/rocm-custom"
rocprofv3 --rocm-root "$ROCM_PATH" -i input.yaml -- "$@"
```

**Environment variable interaction:**

The `--rocm-root` option overrides automatic path detection and environment variables like `ROCM_PATH`:

```
# --rocm-root takes precedence over environment variables
export ROCM_PATH="/opt/rocm-default"
rocprofv3 --rocm-root /opt/rocm-override --hip-trace -- ./app
# Uses /opt/rocm-override, not /opt/rocm-default
```

**Validation and troubleshooting:**

- **Path validation:** Ensure the specified path contains a valid ROCm installation
- **Library compatibility:** Verify that the ROCm version is compatible with your application
- **Permission issues:** Check read permissions for the ROCm directory
- **Path format:** Use absolute paths to avoid ambiguity

## 4.6.5 SDK shared object version control

The `--sdk-soversion` option allows you to specify the shared object version number for the ROCProfiler SDK library. This provides precise control over which version of the library is loaded, useful for testing, compatibility verification, or working with specific library versions.

Shared object versioning follows the Linux convention where libraries have version suffixes like `.so.X` where `X` is the major version number. This option helps resolve library paths when multiple versions are installed.

**Basic usage:**

```
rocprofv3 --sdk-soversion 2 --hip-trace --output-format csv -- <application_path>
```

The preceding command uses `librocprofiler-sdk.so.2` instead of the default version.

**Example scenarios:****Scenario 1: Testing with specific library version**

```
# Test application with SDK version 1
rocprofv3 --sdk-soversion 1 --kernel-trace --pmc SQ_WAVES -- ./app_v1_test

# Test same application with SDK version 2
rocprofv3 --sdk-soversion 2 --kernel-trace --pmc SQ_WAVES -- ./app_v2_test
```

**Scenario 2: Compatibility verification**

```
# Verify backward compatibility with older SDK
rocprofv3 --sdk-soversion 0 --sys-trace -- ./legacy_application
```

### Scenario 3: Development and testing

```
# Use specific version for regression testing
rocprofv3 --sdk-soversion 3 --runtime-trace --output-directory regression_test -- ./test_
↪suite
```

### Scenario 4: Production environment pinning

```
# Pin to specific version for production consistency
rocprofv3 --sdk-soversion 1 --hip-trace --minimum-output-data 100 -- ./production_app
```

### Library resolution behavior:

The option affects library loading in the following order:

1. `librocprofiler-sdk.so.X` (where X is the specified soversion)
2. Fallback to default library if specific version not found

### Using with scripts:

```
#!/bin/bash
# test_matrix.sh - Test with multiple SDK versions
for version in 0 1 2; do
    echo "Testing with SDK SO version $version"
    rocprofv3 --sdk-soversion $version --hip-trace -- ./test_app
done
```

### Troubleshooting:

- **Library not found:** Verify the specified soversion exists in the library path
- **ABI compatibility:** Ensure the SDK version is compatible with your ROCm installation
- **Symbol mismatches:** Check for symbol compatibility between versions
- **Performance differences:** Different versions may have performance characteristics

## 4.6.6 SDK version specification

The `--sdk-version` option allows you to specify the exact version number for the ROCProfiler SDK library resolution. This provides the finest level of control over library selection, useful for testing specific versions, development workflows, or ensuring reproducible profiling environments.

This option helps resolve library paths for version-specific libraries like `librocprofiler-sdk.so.X.Y.Z` where X.Y.Z represents the full semantic version.

### Basic usage:

```
rocprofv3 --sdk-version 1.2.3 --hip-trace --output-format csv -- <application_path>
```

The preceding command uses `librocprofiler-sdk.so.1.2.3` if available.

### Example scenarios:

#### Scenario 1: Exact version testing

```
# Test with specific patch version for bug verification
rocprofv3 --sdk-version 2.1.5 --kernel-trace -- ./bug_reproduction_case

# Test with fixed version
rocprofv3 --sdk-version 2.1.6 --kernel-trace -- ./bug_verification_case
```

## Scenario 2: Reproducible profiling

```
# Ensure exact same SDK version for reproducible results
rocprofv3 --sdk-version 2.2.1 --pmc SQ_WAVES GRBM_COUNT --output-format pftace -- ./
↳ benchmark_app
```

### Version format support:

The option supports various version formats:

- **Semantic versioning:** 1.2.3, 2.0.0, 1.5.10

### Library resolution priority:

When `--sdk-version` is specified, the library resolution follows this order:

1. `librocprofiler-sdk.so.X.Y.Z` (exact version match)
2. `librocprofiler-sdk.so.X.Y` (major.minor match)
3. `librocprofiler-sdk.so.X` (major version match)
4. Default library (`librocprofiler-sdk.so`)

### Using with input files:

While typically used from command line, it can be scripted:

```
#!/bin/bash
# version_matrix_test.sh
VERSIONS=("2.1.0" "2.1.1" "2.1.2" "2.2.0")

for version in "${VERSIONS[@]}; do
    echo "Testing SDK version $version"
    rocprofv3 --sdk-version "$version" --hip-trace --output-directory "results_$version"
↳ -- ./test_app
done
```

### Combined with other version options:

```
# Combine with soversion for maximum control
rocprofv3 --sdk-version 2.1.5 --sdk-soversion 2 --hip-trace -- ./app

# Combine with custom ROCm root
rocprofv3 --rocm-root /opt/rocm-6.0 --sdk-version 2.2.0 --sys-trace -- ./app
```

### Environment integration:

```
# Use environment variable for version
export ROCPROF_SDK_VERSION="2.1.3"
rocprofv3 --sdk-version "$ROCPROF_SDK_VERSION" --kernel-trace -- ./app
```

## 4.6.7 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` or `-A {absolute,relative,type-relative}` 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 output 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
->","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*,
```

(continues on next page)

(continued from previous page)

```
↪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>
```

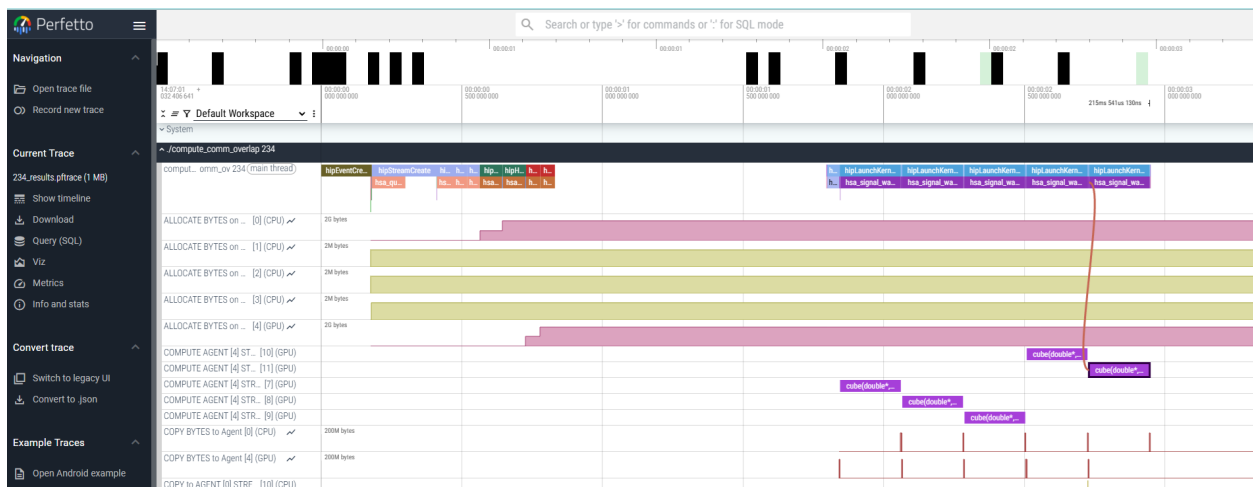
Here is the generated output file with Agent\_Id as "GPU 3":

```
$ 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
```

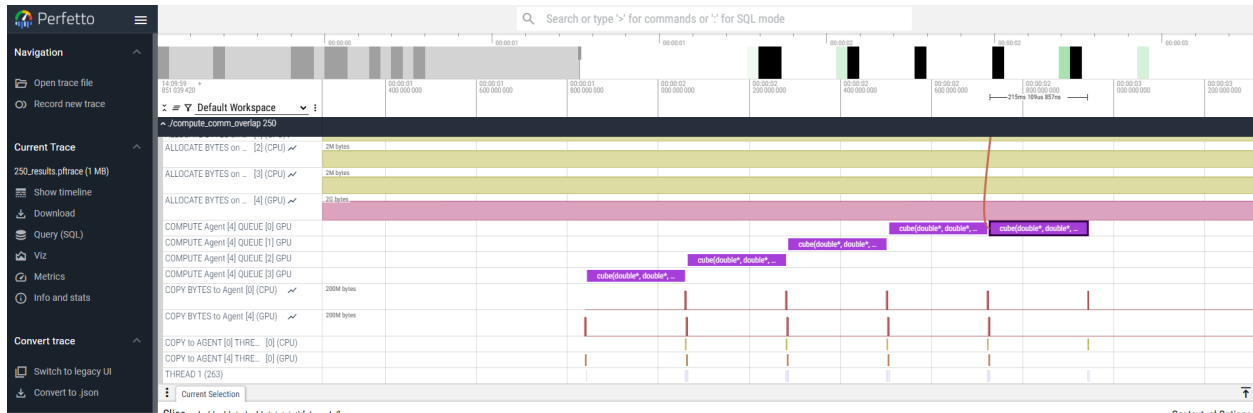
## 4.6.8 Group by queue

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 pfttrace -- <application_path>
```

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



## 4.7 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.

### 4.7.1 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)



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
```

#### 4.7.4 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
```

## 4.8 I/O control options

rocprofv3 provides the following options to control the output.

### 4.8.1 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%

### 4.8.2 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
```

### 4.8.3 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.

### 4.8.4 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>
```

## 4.8.5 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.

## 4.9 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 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_Time	Begin time in nanoseconds (ns) when the kernel begins execution.
End_Time	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_Segment	The amount of memory required in bytes for the combined private, spill, and arg segments for a work item.
Group_Segment	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	The total number of work-items (or, threads) in each workgroup (or, block) launched as part of the kernel dispatch. In HIP, this is equivalent to the total block size.
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	The total number of work-items (or, threads) launched as a part of the kernel dispatch. In HIP, this is equivalent to the total grid size multiplied by the total workgroup (or, block) size.
Grid_Size_n	Number of work-items (or, threads) in the nth dimension required to launch the kernel, where n = X, Y, or Z.
LDS_Block_Size	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_Count	Kernel's Accumulation Vector General Purpose Register (Accum_VGPR/AGPR) count.

## 4.10 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](https://ui.perfetto.dev).

For OTF2 trace visualization, open the trace in [vampir.eu](https://vampir.eu) or any supported visualizer.

**Note**

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

### 4.10.1 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.

#### 4.10.1.1 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 processor.

**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.

## 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 1: `rocprofv3-avail` options

Purpose	Option	Description
avail-options commands	<code>info</code> <code>list</code> <code>pmc-check</code>	Info options for detailed information of counters, agents, and pc-sampling configurations. List options for hardware counters, agents and pc-sampling support. Checking if a set of counters can be collected together on agent.

```
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. Following is the sample output

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

The preceding command generates an output listing counters for all agents, if used with `-d`, only counters on the `-d` device is listed. Output contains following information: logical node id, name and list of PMC counters supported on the agent.

```
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.

```
rocprofv3-avail info
```

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

```
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. Output includes the following information: logical node id, name, counter\_name, description of the counter, dimensions, block/expression for every counter.

```
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. Output has following information: logical node id, method supported, unit, minimum sampling interval, maximum sampling interval flags.

```
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

All commands writes to the standard output.

## DYNAMIC PROCESS ATTACHMENT USING ROCPROFV3

For profiling long-running applications or services where restarting the application is not feasible, `rocprofv3` provides dynamic process attachment using the `--attach` option. This feature facilitates attaching the profiler to a running application without the need to restart it. The attachment is performed using the `ptrace` system call, which enables the profiler to monitor and collect performance data from the target process.

Here is an example syntax for dynamic process attachment:

```
rocprofv3 --attach <PID> [--hip-trace] [--output-format <format>]
```

Here are the options used in the preceding example:

- `<PID>`: Process ID of the target application.
- `--hip-trace`: This optional flag enables HIP API tracing.
- `--output-format`: The desired output format such as `rocpd`, `csv`, or `json`.

### Basic attachment syntax:

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

## 6.1 Basic dynamic process attachment

Follow these steps to attach the profiler to a running process and profile:

1. Start the target application in the background:

```
./myapp -n 1 &
```

2. Get the process ID (PID) of the running application:

```
echo $(pgrep myapp)  
OR  
ps aux | grep myapp
```

3. Attach `rocprofv3` to the running application:

```
rocprofv3 --attach <PID> --hip-trace --output-format rocpd
```

4. Detach the profiler when done:

To detach the profiler from the target application, press “Enter” in the terminal where `rocprofv3` is running. You can also send SIGINT (Ctrl+C) to `rocprofv3` to detach from the target.

5. The profiling data will be saved in the format specified using `output-format`.

## 6.2 Duration-specific dynamic process attachment

Follow these steps to attach the profiler to a running process and profile for a specific duration such as 5 seconds:

1. Start the target application in the background:

```
./myapp -n 1 &
```

2. Get the process ID (PID) of the running application:

```
echo $(pgrep myapp)  
OR  
ps aux | grep myapp
```

3. Attach `rocprofv3` to the running application:

```
rocprofv3 --attach <PID> --attach-duration-msec 5000 --sys-trace --output-format csv
```

4. The profiler will automatically detach after the specified duration (5 seconds in this case). Note that the duration is to be specified in milliseconds (ms).
5. The profiling data will be saved in the format specified using `output-format`. For example, if you specify `--output-format csv`, the data will be saved as a CSV file.

## 6.3 Dynamic process attachment with counter collection

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

The following example attaches the profiler to a process with PID “12345”, collects counters `SQ_WAVES` and `GRBM_COUNT`, and saves profiling data in a CSV file:

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

## 6.4 Key considerations

Here are some important points to be noted while using dynamic process attachment:

- The target process must be running and actively using GPU resources for meaningful profiling data.
- Attachment requires appropriate system permissions. It might even need elevated privileges depending on the target process.
- To use attachment in a docker container, add the `ptrace` capability to the container (`SYS_PTRACE`).
- The profiler collects data for the entire remaining lifetime of the process or until the configured collection period expires. To learn how to configure the collection period, see [Duration-specific dynamic process attachment](#).

## USING ROCPD OUTPUT FORMAT

To accommodate diverse analysis workflows, `rocprofv3` provides comprehensive support for multiple output formats:

- **rocpd** (SQLite3 database) - Default format providing structured data storage.
- **CSV** (Comma-Separated Values) - Tabular format for spreadsheet applications and data analysis tools.
- **JSON** (JavaScript Object Notation) - Structured format optimized for programmatic analysis and integration.
- **PFTrace** (Perfetto protocol buffers) - Binary trace format for high-performance visualization using Perfetto.
- **OTF2** (Open Trace Format 2) - Standardized trace format for interoperability with third-party analysis tools.

The `rocpd` output format serves as the primary data repository for `rocprofv3` profiling sessions. This format leverages SQLite3's ACID-compliant database engine to provide robust, structured storage of comprehensive profiling datasets. The relational schema enables efficient querying and manipulation of profiling data through standard SQL interfaces, facilitating complex analytical operations and custom reporting workflows.

### 7.1 Features

- **Comprehensive data model:** Consolidates all profiling artifacts including execution traces, performance counters, hardware metrics, and contextual metadata within a single SQLite3 database file (`.db` extension).
- **Standards-compliant access:** Supports querying through industry-standard SQL interfaces including command-line tools (`sqlite3` CLI), programming language bindings (Python `sqlite3` module, C/C++ SQLite API), and database management applications.
- **Advanced analytics integration:** Facilitates sophisticated post-processing workflows through custom analytical scripts, automated reporting systems, and integration with third-party visualization and analysis frameworks that provide SQLite3 connectivity.

### 7.2 Generating rocpd output

To generate profiling data in the default `rocpd` format, use:

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

Or, explicitly specify the `rocpd` output format using the `--output-format` parameter:

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

The profiling session generates output files following the naming convention `%hostname%/%pid%_results.db`, where:

- `%hostname%`: The system hostname.

- %pid%: The process identifier of the profiled application.

## 7.3 Converting rocpd to alternative formats

The rocpd database format supports conversion to alternative output formats for specialized analysis and visualization workflows.

The rocpd conversion utility is distributed as part of the ROCm installation package, located in `/opt/rocm-<version>/bin`, and provides both executable and Python module interfaces for programmatic integration.

To transform database files into target formats, run the `rocpd convert` command with appropriate parameters.

- **CSV format conversion**

```
/opt/rocm/bin/rocpd convert -i <input-file>.db --output-format csv
```

The converted CSV files are generated as `rocpd-output-data/out_hip_api_trace.csv`, where the `rocpd-output-data` is relative to the current working directory.

- **OTF2 format conversion**

```
/opt/rocm/bin/rocpd convert -i <input-file>.db --output-format otf2
```

- **Perfetto trace format conversion**

```
/opt/rocm/bin/rocpd convert -i <input-file>.db --output-format pftrace
```

### Python interpreter compatibility:

On encountering Python interpreter version conflicts, specify the appropriate Python executable explicitly:

```
python3.10 $(which rocpd) convert -f csv -i <input-file>.db
```

### 7.3.1 rocpd convert command-line options

The command-line options as displayed using `rocpd convert --help` are listed here:

```
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]
```

The following table provides a detailed listing of the rocpd convert command-line options:

Category	Option	Description
Required arguments	-i INPUT [INPUT ...], --input INPUT [INPUT ...] -f {csv,pftrace,otf2} [{csv,pftrace,otf2} ...], --output-format {csv,pftrace,otf2} [{csv,pftrace,otf2} ...]	Specifies input database file paths. Accepts multiple SQLite3 database files separated by whitespace for batch processing operations. Defines target output formats. Supports concurrent conversion to multiple formats such as CSV, PFTrace, and OTF2.
I/O configuration	-o OUTPUT_FILE, --output-file OUTPUT_FILE -d OUTPUT_PATH, --output-path OUTPUT_PATH	Configures the base filename for generated output files (default: out). Specifies the target directory for output file generation (default: ./rocpd-output-data).
Kernel identification Options	--kernel-rename	Substitutes kernel function names with the corresponding ROCTX marker annotations for enhanced semantic context.
Device identification configuration	--agent-index-value {absolute,relative,type-relative}	Controls device identification methodology in the converted output. Here are the values: <ul style="list-style-type: none"> <li>• <b>absolute</b>: Utilizes hardware node identifiers such as Agent-0, Agent-2, and Agent-4, while bypassing container group abstractions.</li> <li>• <b>relative</b>: Employs logical node identifiers such as Agent-0, Agent-1, and Agent-2, while incorporating container group context. This is the Default value.</li> <li>• <b>type-relative</b>: Applies device-type-specific logical identifiers such as CPU-0, GPU-0, and GPU-1, with independent numbering sequence per device class.</li> </ul>
Perfetto trace configuration	--perfetto-backend {inprocess,system} --perfetto-buffer-fill-policy {discard,ring_buffer} --perfetto-buffer-size KB --perfetto-shmem-size-hint KB --group-by-queue	Configures Perfetto data collection architecture. The value <b>system</b> requires active traced and <b>perfetto</b> daemon processes, while <b>inprocess</b> operates autonomously. The default value is <b>inprocess</b> . Defines buffer overflow handling strategy. The value <b>discard</b> drops new records when capacity is exceeded and <b>ring_buffer</b> overwrites oldest records. The

### 7.3. Converting rocpd to alternative formats

## 7.4 Types of conversion

Here are the types of conversion supported by rocpd:

- Single database conversion to Perfetto format

```
/opt/rocm/bin/rocpd convert -i db1.db --output-format pfttrace
```

- Multi-Database conversion with temporal filtering

The following example converts multiple databases to Perfetto format while specifying custom output directory and filename with temporal window constraint set to the final 70% of the trace duration:

```
/opt/rocm/bin/rocpd convert -i db1.db db2.db --output-format pfttrace -d "./output/"
↪ -o "twoFileTraces" --start 30% --end 100%
```

- Batch conversion into multiple formats

The following example processes six database files simultaneously, generating both CSV and Perfetto trace outputs with custom output configuration:

```
/opt/rocm/bin/rocpd convert -i db{0..5}.db --output-format csv pfttrace -d "~/output_
↪ folder/" -o "sixFileTraces"
```

- Comprehensive format conversion

The following example converts multiple databases into all supported formats (CSV, OTF2, and Perfetto trace) in a single operation:

```
/opt/rocm/bin/rocpd convert -i db{3,4}.db --output-format csv otf2 pfttrace
```

## 7.5 Dedicated conversion tools

ROCprofiler-SDK provides specialized conversion utilities for efficient format-specific operations. The following tools offer streamlined interfaces for single-format conversions, which are particularly useful in automated workflows and scripts.

### 7.5.1 rocpd2csv - CSV export tool

**Purpose:** To convert rocpd SQLite3 databases to CSV format for spreadsheet analysis and data processing workflows.

**Location:** /opt/rocm/bin/rocpd2csv

**Syntax:**

```
rocpd2csv -i INPUT [INPUT ...] [OPTIONS]
```

**Key features:**

- Structured data export: Converts hierarchical database content to tabular CSV format.
- Multidatabase support: Aggregates data from multiple database files into a unified CSV output.
- Time window filtering: Applies temporal filters to limit exported data range.
- Configurable output: Facilitates customized output file naming and directory structure.

**Usage examples:**

```
# Basic CSV conversion
rocpd2csv -i profile_data.db

# Convert multiple databases with custom output path
rocpd2csv -i db1.db db2.db db3.db -d ~/analysis_output/ -o combined_profile

# Apply time window filtering (export middle 70% of execution)
rocpd2csv -i large_profile.db --start 15% --end 85%
```

**Common output files:**

- out\_hip\_api\_trace.csv: HIP API call trace data.
- out\_kernel\_trace.csv: GPU kernel execution information.
- out\_counter\_collection.csv: Hardware performance counter data.

**7.5.2 rocpd2otf2 - OTF2 export tool**

**Purpose:** To generate OTF2 files for high-performance trace analysis using tools such as Vampir, Tau, and Score-P viewers.

**Location:** /opt/rocm/bin/rocpd2otf2

**Syntax:**

```
rocpd2otf2 -i INPUT [INPUT ...] [OPTIONS]
```

**Key features:**

- HPC-standard format: Produces traces compatible with scientific computing analysis tools.
- Hierarchical timeline: Preserves process, thread, or queue relationships in trace structure.
- Scalable storage: Efficient binary format for large-scale profiling data.
- Agent indexing: Configurable GPU agent indexing strategies (absolute, relative, or type-relative).

**Usage examples:**

```
# Generate OTF2 trace archive
rocpd2otf2 -i gpu_workload.db

# Multi-process trace with custom indexing
rocpd2otf2 -i mpi_rank_*.db --agent-index-value type-relative -o mpi_trace

# Time-windowed trace export
rocpd2otf2 -i long_execution.db --start-marker "computation_begin" --end-marker
↔ "computation_end"
```

**Output structure:**

- trace.otf2: Main trace archive containing timeline data.
- trace.def: Trace definition file with metadata.
- Supporting files for multistream trace data.

### 7.5.3 rocpd2pftrace - Perfetto trace export

**Purpose:** To convert rocpd databases to Perfetto protocol buffer format for interactive visualization using the [Perfetto UI](#).

**Location:** /opt/rocm/bin/rocpd2pftrace

**Syntax:**

```
rocpd2pftrace -i INPUT [INPUT ...] [OPTIONS]
```

**Key features:**

- Interactive visualization: Optimized for modern web-based trace viewers.
- Real-time analysis: Supports streaming analysis workflows.
- GPU timeline integration: Specialized visualization of GPU execution patterns.
- Configurable backend: Supports both in-process and system-wide tracing backends.

**Backend configuration options:**

```
# In-process backend (default)
rocpd2pftrace -i profile.db --perfetto-backend inprocess

# System-wide tracing backend
rocpd2pftrace -i system_profile.db --perfetto-backend system \
    --perfetto-buffer-size 64MB --perfetto-shmem-size-hint 32MB
```

**Buffer management options:**

```
# Ring buffer mode (overwrites old data)
rocpd2pftrace -i continuous_profile.db --perfetto-buffer-fill-policy ring_buffer

# Discard mode (stops recording when full)
rocpd2pftrace -i bounded_profile.db --perfetto-buffer-fill-policy discard
```

**Usage examples:**

```
# Basic Perfetto trace generation
rocpd2pftrace -i application.db

# High-throughput configuration
rocpd2pftrace -i heavy_workload.db --perfetto-buffer-size 128MB \
    --perfetto-buffer-fill-policy ring_buffer

# Multi-queue analysis
rocpd2pftrace -i multi_stream.db --group-by-queue -o queue_analysis
```

**Visualization workflow:**

Follow these steps to visualize traces on Perfetto:

1. Generate .perfetto-trace file using rocpd2pftrace.
2. Open <https://ui.perfetto.dev> in web browser.
3. Load generated trace file for interactive analysis.

## 7.5.4 rocpd2summary - statistical analysis tool

**Purpose:** To generate comprehensive statistical summaries and performance analysis reports from rocpd profiling data.

**Location:** /opt/rocm/bin/rocpd2summary

**Syntax:**

```
rocpd2summary -i INPUT [INPUT ...] [OPTIONS]
```

**Key features:**

- Multiformat output: Supports console, CSV, HTML, JSON, Markdown, and PDF report generation.
- Comprehensive statistics: Statistics include kernel execution times, API call frequencies, and memory transfer analysis.
- Domain-specific analysis: Generates separate summaries for HIP, ROCr, Markers, and other trace domains. For examples, see analysis-categories.
- Rank-based analysis: Per-process and per-rank performance breakdowns for MPI applications.
- Configurable scope: Selective inclusion or exclusion of analysis categories.

**Output format options:**

```
# Console output (default)
rocpd2summary -i profile.db

# CSV format for data analysis
rocpd2summary -i profile.db --format csv -o performance_metrics

# HTML report with visualization
rocpd2summary -i profile.db --format html -d ~/reports/

# Multiple output formats
rocpd2summary -i profile.db --format csv html json
```

**Analysis categories:**

Here is how you can generate summary for specific trace domains:

```
# Include all available domains
rocpd2summary -i profile.db --region-categories HIP HSA MARKERS KERNEL

# Focus on GPU kernel analysis only
rocpd2summary -i profile.db --region-categories KERNEL

# Exclude markers to speed up processing
rocpd2summary -i profile.db --region-categories HIP HSA KERNEL
```

**Advanced analysis options:**

The following commands demonstrate the usage of advanced analysis options such as domain-summary, summary-by-rank, and start/end:

```
# Include domain-specific statistics
rocpd2summary -i multi_gpu.db --domain-summary
```

(continues on next page)

```
# Per-rank analysis for MPI applications
rocpd2summary -i mpi_profile_*.db --summary-by-rank --format html
```

```
# Time-windowed summary analysis
rocpd2summary -i long_run.db --start 25% --end 75% --format csv
```

**Report content:**

- Kernel statistics: Execution time distributions, call frequencies, and grid or block sizes.
- API timing: HIP or HSA function call durations and frequencies.
- Memory analysis: Transfer patterns, bandwidth utilization, and allocation statistics.
- Device utilization: GPU occupancy patterns and idle time analysis.
- Synchronization overhead: Barrier and synchronization point analysis.

**Output files:**

- `kernels_summary.{format}` - GPU kernel execution summary.
- `hip_summary.{format}` - HIP API call statistics.
- `hsa_summary.{format}` - HSA runtime API analysis.
- `memory_summary.{format}` - Memory operation statistics.
- `markers_summary.{format}` - Marker event analysis.

## 7.6 Generating performance summary using rocpd

The `rocpd summary` command provides statistical analysis and performance summaries, similar to the summary functionality available in `rocprofv3`. This command generates comprehensive reports from `rocpd` database files, offering the same analytical capabilities that were previously available through `rocprofv3 --summary` but on the structured database format.

**Purpose:** To generate statistical summaries and performance reports from `rocpd` database files, providing similar functionality as `rocprofv3` builtin summary capabilities.

**Location:** `/opt/rocm/bin/rocpd summary`

**Syntax:**

```
rocpd summary -i INPUT [INPUT ...] [OPTIONS]
```

**Key features:**

- Compatible analysis: Provides the same summary statistics and reports as `rocprofv3 --summary`.
- Database-driven: Operates on structured `rocpd` database files for consistent and reproducible analysis.
- Multidatabase aggregation: Combines and analyzes data from multiple profiling sessions, ranks, or nodes in a single operation. This is further explained in *Multidatabase summary analysis*.
- Comparative analysis: Uses `--summary-by-rank` to compare performance across different ranks, nodes, or execution contexts.
- Flexible output: Generates summaries in multiple formats such as console, CSV, HTML, and JSON.
- Selective reporting: Allows generating reports based on specific performance domains and categories.

## 7.6.1 Multidatabase summary analysis

The `rocpd summary` command excels at aggregating multiple database files, providing capabilities not available with single-session analysis.

Here are the benefits of using `rocpd summary` for multidatabase summary:

- **Unified summary reports:**

```
# Aggregate multiple databases into single comprehensive summary
rocpd summary -i session1.db session2.db session3.db --format html -o unified_
↳summary

# Combine all MPI rank databases for overall application analysis
rocpd summary -i rank_*.db --format csv -o mpi_application_summary

# Time-series aggregation across multiple profiling runs
rocpd summary -i daily_profile_*.db --format json -o weekly_performance_trends
```

- **Rankwise comparative analysis:**

The `--summary-by-rank` option provides detailed comparative analysis, helping you to identify performance variations, load balancing issues, and optimization opportunities across different execution contexts:

```
# Compare performance across MPI ranks
rocpd summary -i rank_0.db rank_1.db rank_2.db rank_3.db --summary-by-rank --format_
↳html -o rank_comparison

# Analyze multi-node performance characteristics
rocpd summary -i node_*.db --summary-by-rank --format csv -o node_performance_
↳analysis

# Compare GPU device performance in multi-GPU applications
rocpd summary -i gpu_0.db gpu_1.db gpu_2.db gpu_3.db --summary-by-rank --format_
↳json -o gpu_scaling_analysis
```

### 7.6.1.1 Use cases for multidatabase summary analysis

- **MPI application performance analysis:**

```
# Profile distributed MPI application
mpirun -np 8 rocprofv3 --hip-trace --output-format rocpd -- mpi_simulation

# Generate unified summary for overall application performance
rocpd summary -i results_rank_*.db --format html -o application_overview

# Identify load balancing issues with rank-by-rank comparison
rocpd summary -i results_rank_*.db --summary-by-rank --format csv -o load_balance_
↳analysis
```

- **Multi-GPU scaling studies:**

```
# Profile scaling from 1 to 4 GPUs
for gpus in 1 2 4; do
    rocprofv3 --hip-trace --device 0:${((gpus-1))} --output-format rocpd -o "scaling_
↳{gpus}gpu.db" -- gpu_benchmark
```

(continues on next page)

(continued from previous page)

**done**

```
# Aggregate scaling analysis
rocpd summary -i scaling_*gpu.db --format html -o gpu_scaling_summary

# Compare efficiency across different GPU counts
rocpd summary -i scaling_*gpu.db --summary-by-rank --format json -o scaling_
↪efficiency
```

- **Performance regression testing:**

```
# Profile baseline and optimized versions
rocprofv3 --hip-trace --output-format rocpd -o baseline.db -- application_v1
rocprofv3 --hip-trace --output-format rocpd -o optimized.db -- application_v2

# Generate unified performance comparison
rocpd summary -i baseline.db optimized.db --summary-by-rank --format html -o ↪
↪regression_analysis
```

- **Cross-platform performance comparison:**

```
# Profile on different hardware platforms
rocprofv3 --hip-trace --output-format rocpd -o platform_A.db -- benchmark
rocprofv3 --hip-trace --output-format rocpd -o platform_B.db -- benchmark

# Compare platform performance characteristics
rocpd summary -i platform_*.db --summary-by-rank --format csv -o platform_comparison
```

- **Domain-specific reporting:**

```
# Cross-rank summary for MPI applications with domain focus
rocpd summary -i rank_*.db --summary-by-rank --region-categories KERNEL HIP --
↪format html
```

- **Time-windowed analysis:**

```
rocpd summary -i profile_*.db --start 25% --end 75% --summary-by-rank
```

- **Domain-specific comparative analysis:**

```
rocpd summary -i node_*.db --domain-summary --summary-by-rank --region-categories ↪
↪HIP ROCR
```

In summary, here is how the output generated using `rocpd summary` enhances performance analysis:

- **Unified summaries:** Provides aggregate statistics across all input databases, showing combined performance metrics.
- **Rankwise summaries:** Generates separate statistical reports for each input database, enabling direct comparison of performance characteristics.
- **Comparative metrics:** Highlights performance variations, identifies outliers, and reveals load balancing opportunities.

## 7.6.2 Integration with rocprofv3 workflow

The `rocpd summary` command maintains full compatibility with `rocprofv3 summary` analysis, while extending capabilities to multidatabase scenarios. Users familiar with `rocprofv3 --summary` will find identical statistical outputs and report formats when using `rocpd summary` on database files, with the added benefit of cross-session analysis capabilities.

## 7.7 Aggregating multiprofilng data using rocpd

One of the key advantages of the `rocpd` format is its ability to aggregate and analyze data from multiple profiling sessions, ranks, or nodes within a unified framework. This capability enables comprehensive analysis workflows, which was not possible with earlier output formats.

Here are the use cases of data aggregation using `rocpd`:

- **Multidatabase analysis capabilities.**

Unlike the Perfetto output format used in earlier versions, `rocpd` databases can be seamlessly combined for cross-session analysis:

```
# Aggregate analysis across multiple profiling sessions
rocpd query -i session1.db session2.db session3.db \
    --query "SELECT name, AVG(duration) FROM kernels GROUP BY name"

# Cross-rank performance comparison for MPI applications
rocpd summary -i rank_0.db rank_1.db rank_2.db rank_3.db --summary-by-rank

# Multi-node scaling analysis
rocpd query -i node_*.db \
    --query "SELECT COUNT(*) as total_kernels, SUM(duration) as total_time_
    ↪FROM kernels"
```

- **Distributed computing workflows.**

`rocpd` can effectively aggregate data in a distributed computing environment. Here are the examples for single and multi-GPU analysis:

- MPI application analysis:

```
# Profile MPI application across multiple ranks
mpirun -np 4 rocprofv3 --hip-trace --output-format rocpd -- mpi_application

# Generate aggregated performance summary
rocpd summary -i results_rank_*.db --summary-by-rank --format html -o mpi_
    ↪performance_report

# Analyze load balancing across ranks
rocpd query -i results_rank_*.db \
    --query "SELECT pid, COUNT(*) as kernel_count, AVG(duration) as_
    ↪avg_duration FROM kernels GROUP BY pid"
```

- Multi-GPU scaling analysis:

```
# Profile application with multiple GPU devices
rocprofv3 --hip-trace --device 0,1,2,3 --output-format rocpd -- multi_gpu_app
```

(continues on next page)

(continued from previous page)

```
# Aggregate device utilization analysis
rocpd query -i multi_gpu_results.db \
    --query "SELECT agent_abs_index as device_id, COUNT(*) as_
↳operations, SUM(duration) as total_time FROM kernels GROUP BY device_id"

# Cross-device performance comparison
rocpd summary -i multi_gpu_results.db --domain-summary
```

- **Temporal aggregation.**

- Time-series analysis:

```
# Collect profiles over time for performance monitoring
for hour in {1..24}; do
    rocprofv3 --hip-trace --output-format rocpd -o "profile_hour_${hour}.db" --_
↳application
done

# Analyze performance trends over time
rocpd query -i profile_hour_*.db \
    --query "SELECT AVG(duration) as avg_kernel_time, COUNT(*) as_
↳kernel_count FROM kernels" \
    --format csv -o performance_trends
```

- Comparative analysis:

```
# Compare baseline vs optimized performance
rocpd query -i baseline.db optimized.db \
    --query "SELECT kernel, AVG(duration) as avg_time FROM kernels_
↳GROUP BY name ORDER BY avg_time DESC"

# Generate comparative summary reports
rocpd summary -i baseline.db optimized.db --format html -o comparison_report
```

Here are the benefits of data aggregation using rocpd:

- **Unified analysis:** Combines data from different execution contexts, hardware configurations, and time periods.
- **Scalability insights:** Analyzes performance scaling across multiple nodes, ranks, or GPU devices.
- **Trend analysis:** Tracks performance evolution over time or across different software versions.
- **Load balancing:** Identifies performance bottlenecks and load distribution issues in distributed applications.
- **Cross-platform comparison:** Compares performance across different hardware platforms using unified database schema.

The aggregation capabilities of rocpd format enable sophisticated analysis workflows that provide deeper insight into application performance characteristics across diverse computing environments.

## 7.8 Tool integration and workflow examples

The following examples demonstrate how to use the *rocpd conversion tools* for automated performance monitoring and multiformat analysis.

- **Multiformat analysis pipeline:**

```
# Generate all analysis formats for comprehensive review
rocpd2csv -i profile.db -o analysis_data
rocpd2summary -i profile.db --format html -o performance_report
rocpd2pftrace -i profile.db -o interactive_trace
```

- **Automated performance monitoring:**

```
#!/bin/bash
# Performance analysis automation script

PROFILE_DB="$1"
OUTPUT_DIR="analysis_$(date +%Y%m%d_%H%M%S)"

mkdir -p "$OUTPUT_DIR"

# Generate CSV data for automated analysis
rocpd2csv -i "$PROFILE_DB" -d "$OUTPUT_DIR" -o raw_data

# Create summary reports
rocpd2summary -i "$PROFILE_DB" --format csv html \
              -d "$OUTPUT_DIR" -o performance_summary

# Generate interactive trace for detailed investigation
rocpd2pftrace -i "$PROFILE_DB" -d "$OUTPUT_DIR" -o interactive_trace
```

## 7.9 Executing SQL queries with rocpd

The `rocpd query` command provides comprehensive SQL-based analysis capabilities for exploring and extracting data from rocpd databases. This command enables custom analysis workflows and automated reporting for GPU profiling data analysis, and integration with external analysis pipelines.

**Purpose:** To execute custom SQL queries against rocpd databases with support for multiple output formats, automated reporting, and Email delivery.

**Location:** `/opt/rocm/bin/rocpd query`

**Syntax:**

```
rocpd query -i INPUT [INPUT ...] --query "SQL_STATEMENT" [OPTIONS]
```

**Key features:**

- Standard SQL support: Supports full SQLite3 SQL syntax including JOINS, aggregate functions, and complex WHERE clauses.
- Multidatabase aggregation: Facilitates queries across multiple database files as a unified virtual database.
- Multiple output formats: Supports output formats such as console, CSV, HTML, JSON, Markdown, PDF, and interactive dashboards.
- Script execution: Can execute complex SQL scripts with view definitions and custom functions.
- Automated reporting: Supports Email delivery with SMTP configuration and attachment management.
- Time window integration: Allows temporal filtering before query execution.

## 7.9.1 Database schema and views

rocpd databases provide the following comprehensive views for analysis. It is generally recommended to build a query using `data_views`:

- **Core data views:**

```
-- System and hardware information
SELECT * FROM rocpd_info_agents;
SELECT * FROM rocpd_info_node;

-- Kernel execution data
SELECT * FROM kernels;
SELECT * FROM top_kernels;

-- API trace information
SELECT * FROM regions_and_samples WHERE category LIKE 'HIP_%';
SELECT * FROM regions_and_samples WHERE category LIKE 'RCCL_%';

-- Performance counters
SELECT * FROM counters_collection;

-- Memory operations
SELECT * FROM memory_copies;
SELECT * FROM memory_allocations;

-- Process and thread information
SELECT * FROM processes;
SELECT * FROM threads;

-- Marker and region data
SELECT * FROM regions;
SELECT * FROM regions_and_samples WHERE category LIKE 'MARKERS_%';
```

- **Summary and analysis views:**

```
-- Top performing kernels by execution time
SELECT * FROM top_kernels LIMIT 10;

-- Top Analysis
SELECT * FROM top;

-- Busy Analysis
SELECT * FROM busy;
```

## 7.9.2 rocpd query examples

The following examples demonstrate the usage of `rocpd query` for performing some useful operations:

- **Simple data exploration:**

```
# List available GPU agents
rocpd query -i profile.db --query "SELECT * FROM rocpd_info_agents"

# Show top 10 longest-running kernels
```

(continues on next page)

(continued from previous page)

```

rocpd query -i profile.db --query "SELECT name, duration FROM kernels ORDER BY
↳duration DESC LIMIT 10"

# Count total number of kernel dispatches
rocpd query -i profile.db --query "SELECT COUNT(*) as total_kernels FROM kernels"

```

- **Multidatabase aggregation:**

```

# Combine data from multiple profiling sessions
rocpd query -i session1.db session2.db session3.db \
    --query "SELECT pid, COUNT(*) as kernel_count FROM kernels GROUP BY pid"

# Cross-session performance comparison
rocpd query -i baseline.db optimized.db \
    --query "SELECT name as kernel_name, AVG(duration) as avg_duration FROM
↳kernels GROUP BY kernel_name"

```

- **Advanced analytics:**

```

# Kernel performance analysis with statistics
rocpd query -i profile.db --query "
SELECT
    name as kernel_name,
    COUNT(*) as dispatch_count,
    MIN(duration) as min_duration,
    AVG(duration) as avg_duration,
    MAX(duration) as max_duration,
    SUM(duration) as total_duration
FROM kernels
GROUP BY kernel_name
ORDER BY total_duration DESC"

```

- **Memory transfer analysis:**

```

# Memory copy analysis by direction
rocpd query -i profile.db --query "
SELECT
    name as kernel_name,
    src_agent_type,
    src_agent_abs_index,
    dst_agent_type,
    dst_agent_abs_index,
    COUNT(*) as transfer_count,
    SUM(size) as total_bytes,
    SUM(duration) as total_duration
FROM memory_copies
GROUP BY src_agent_abs_index
ORDER BY total_bytes DESC"

```

### 7.9.3 Output format options

rocpcd query supports the following output formats:

- **Console output (default):**

```
# Display results in terminal
rocpcd query -i profile.db --query "SELECT * FROM top_kernels LIMIT 5"
```

- **CSV export for data analysis:**

```
# Export to CSV file
rocpcd query -i profile.db --query "SELECT * FROM kernels" --format csv -o kernel_
↪analysis

# Specify custom output directory
rocpcd query -i profile.db --query "SELECT * FROM kernels" --format csv -d ~/
↪analysis/ -o kernel_data
```

- **HTML reports:**

```
# Generate HTML table
rocpcd query -i profile.db --query "SELECT * FROM top_kernels" --format html -o_
↪performance_report
```

- **Interactive dashboard:**

```
# Create interactive HTML dashboard
rocpcd query -i profile.db --query "SELECT * FROM device_utilization" --format_
↪dashboard -o utilization_dashboard

# Use custom dashboard template
rocpcd query -i profile.db --query "SELECT * FROM kernels" --format dashboard \
--template-path ~/templates/custom_dashboard.html -o custom_report
```

- **JSON for programmatic integration:**

```
# Export structured JSON data
rocpcd query -i profile.db --query "SELECT * FROM counters_collection" --format json_
↪-o counter_data
```

- **PDF reports:**

```
# Generate PDF report with monospace formatting
rocpcd query -i profile.db --query "SELECT name, duration FROM top_kernels" --format_
↪pdf -o kernel_report
```

### 7.9.4 Script-based analysis

You can use rocpcd query to execute complex SQL scripts with view definitions and custom analysis logic.

Here is a sample SQL script (analysis.sql):

```
-- Create temporary views for complex analysis
CREATE TEMP VIEW kernel_stats AS
SELECT
```

(continues on next page)

(continued from previous page)

```

name as kernel_name,
COUNT(*) as dispatch_count,
AVG(duration) as avg_duration,
STDDEV(duration) as duration_stddev
FROM kernels
GROUP BY kernel_name;

CREATE TEMP VIEW performance_outliers AS
SELECT k.*, ks.avg_duration, ks.duration_stddev
FROM kernels k
JOIN kernel_stats ks ON k.name = ks.name
WHERE ABS(k.duration - ks.avg_duration) > 2 * ks.duration_stddev;

```

Here is how to execute the preceding script using `rocpd query`:

```

# Run script then execute query
rocpd query -i profile.db --script analysis.sql \
    --query "SELECT * FROM performance_outliers" --format html -o outlier_
↪analysis

```

## 7.9.5 Time window integration

Here is how to apply temporal filtering before query execution:

```

# Query only middle 50% of execution timeline
rocpd query -i profile.db --start 25% --end 75% \
    --query "SELECT COUNT(*) as kernel_count FROM kernels"

# Use marker-based time windows
rocpd query -i profile.db --start-marker "computation_begin" --end-marker "computation_
↪end" \
    --query "SELECT * FROM kernels ORDER BY start_time"

# Absolute timestamp filtering
rocpd query -i profile.db --start 1000000000 --end 2000000000 \
    --query "SELECT * FROM kernels WHERE start_time BETWEEN 1000000000 AND_
↪2000000000"

```

## 7.9.6 Automated Email reporting

`rocpd query` facilitates sending reports through Email. Here is how you can send reports in different formats and use various Email configuration options:

- **Basic Email delivery:**

```

# Send CSV report via email
rocpd query -i profile.db --query "SELECT * FROM top_kernels" --format csv \
    --email-to analyst@company.com --email-from profiler@company.com \
    --email-subject "Weekly Performance Report"

```

- **Advanced Email configuration:**

```
# Multiple recipients with SMTP authentication
rocpd query -i profile.db --query "SELECT * FROM device_utilization" --format html \
  --email-to "team@company.com,manager@company.com" \
  --email-from profiler@company.com \
  --email-subject "GPU Utilization Analysis" \
  --smtp-server smtp.company.com --smtp-port 587 \
  --smtp-user profiler@company.com --smtp-password $(cat ~/.smtp_pass) \
  --inline-preview --zip-attachments
```

- **Dashboard Email reports:**

```
# Send interactive dashboard via email
rocpd query -i profile.db --query "SELECT * FROM kernels" --format dashboard \
  --template-path ~/templates/executive_summary.html \
  --email-to executives@company.com --email-from profiler@company.com \
  --email-subject "Executive Performance Dashboard" \
  --inline-preview
```

## 7.9.7 Integration workflows

- **Automated analysis pipeline:**

```
#!/bin/bash
# Automated reporting script

DB_FILE="$1"
REPORT_DATE=$(date +%Y-%m-%d)

# Generate multiple analysis reports
rocpd query -i "$DB_FILE" --query "SELECT * FROM top_kernels LIMIT 20" \
  --format html -o "top_kernels_$REPORT_DATE"

rocpd query -i "$DB_FILE" --query "SELECT * FROM memory_copy_summary" \
  --format csv -o "memory_analysis_$REPORT_DATE"

rocpd query -i "$DB_FILE" --query "SELECT * FROM device_utilization" \
  --format dashboard -o "utilization_dashboard_$REPORT_DATE" \
  --email-to team@company.com --email-from automation@company.com
```

- **Performance regression detection:**

```
# Compare current performance against baseline
rocpd query -i baseline.db current.db --script performance_comparison.sql \
  --query "SELECT * FROM performance_regression_analysis" \
  --format html -o regression_report \
  --email-to devteam@company.com --email-from ci@company.com \
  --email-subject "Performance Regression Analysis"
```

- **Custom analysis functions:**

rocpd databases support custom SQL functions for advanced analysis:

```
# Use built-in rocpd functions
rocpd query -i profile.db --query "
```

(continues on next page)

(continued from previous page)

```
SELECT
  name,
  rocpd_get_string(name_id, 0, nid, pid) as full_kernel_name,
  duration
FROM kernels
WHERE rocpd_get_string(name_id, 0, nid, pid) LIKE '%gemm%'
```

### 7.9.8 rocpd query command-line reference

The command-line options as displayed using `rocpd query --help` are listed here:

```
usage: rocpd query [-h] -i INPUT [INPUT ...] --query QUERY [--script SCRIPT]
                  [--format {console, csv, html, json, md, pdf, dashboard, clipboard}]
                  [-o OUTPUT_FILE] [-d OUTPUT_PATH]
                  [--email-to EMAIL_TO] [--email-from EMAIL_FROM]
                  [--email-subject EMAIL_SUBJECT] [--smtp-server SMTP_SERVER]
                  [--smtp-port SMTP_PORT] [--smtp-user SMTP_USER]
                  [--smtp-password SMTP_PASSWORD] [--zip-attachments]
                  [--inline-preview] [--template-path TEMPLATE_PATH]
                  [--start START | --start-marker START_MARKER]
                  [--end END | --end-marker END_MARKER]
```

The following table provides a detailed listing of the `rocpd query` command-line options:

Category	Option	Description
Required arguments	<pre>-i INPUT [INPUT ...], --input INPUT [INPUT ...] --query QUERY</pre>	<p>The input database file paths. Multiple databases are merged into a unified view.</p> <p>The SQL SELECT statement to be executed. Enclose complex queries in quotes.</p>
Query options	<pre>--script SCRIPT --format {console, csv, html, json, md, pdf, dashboard, clipboard}</pre>	<p>The SQL script file to be executed before running the main query. Useful for creating views and functions.</p> <p>The output format. Dashboard format creates interactive HTML reports. The default value is console.</p>
Output configuration	<pre>-o OUTPUT_FILE, --output-file OUTPUT_FILE -d OUTPUT_PATH, --output-path OUTPUT_PATH --template-path TEMPLATE_PATH</pre>	<p>Base filename for exported files. Output directory path. Jinja2 template file for dashboard format customization.</p>
Email reporting	<pre>--email-to EMAIL_TO --email-from EMAIL_FROM --email-subject EMAIL_SUBJECT --smtp-server SMTP_SERVER, --smtp-port SMTP_PORT --smtp-user SMTP_USER, --smtp-password SMTP_PASSWORD --zip-attachments --inline-preview</pre>	<p>Recipient Email addresses (comma-separated for multiple recipients). Sender Email address. This is required when using Email delivery. Email subject line. SMTP server configuration. The default value is localhost:25. SMTP authentication credentials. Bundles all attachments into a single ZIP file. Embeds HTML reports as Email body content.</p>
Time window filtering	<pre>--start START, --end END --start-marker START_MARKER, --end-marker END_MARKER</pre>	<p>Temporal boundaries using percentage such as 25% or absolute timestamps. Named marker events defining time window boundaries.</p>

## 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.

### 8.1 ROCTX annotations

ROCTX provides two types of annotations: markers and ranges.

#### 8.1.1 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.

#### 8.1.2 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.

#### 8.1.3 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.

## 8.2 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 1: Marker api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
MARKER_	before hipLaunchF ernel	717	717	1	1520113899312225	1520113899312225
MARKER_	after hipLaunchF ernel	717	717	4	1520113900128482	1520113900128482
MARKER_	hipMem- cpy	717	717	5	1520113900141100	1520113901483408
MARKER_	hipLaunchF ernel	717	717	3	1520113899684965	1520113901491622
MARKER_	hipLaunchF ernel 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
```

### 8.3 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 2: resource naming

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

## 8.4 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 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_transpc	15964	15964	6	1392141765525156	1392141765532696
MARKER_	run(rows=4 cols=5)	15964	15964	3	1392141765498106	1392141765534186
MARKER_	main	15964	15964	2	1392141765494795	1392141765574506

## 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
```

## 9.1 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
```

## 9.2 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
```

## 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.

### 10.1 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;
}
```

## 10.2 Building the OpenMP Offload Application

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

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

## 10.3 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.

## 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.

## 11.1 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.

```
GPU:0
NAME:gfx90a
configs:
  Method      :host_trap
  Unit        :time
  Min_Interval :1
  Max_Interval :18446744073709551615
  Flags       :none
```

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 1: PC sampling host trap

Sample_Timestamp	Exec_Mask	Dis-patch_Id	Instruction	Instruc-tion_Comm	Correlation_Id
3464444413017201	65535	1	s_endpgm		1
3464444413017201	65535	1	s_waitcnt vncnt(0)		1
3464444413018481	65535	1	s_waitcnt vncnt(0)		1
3464444413018481	65535	1	s_endpgm		1
3464444413018481	65535	1	s_waitcnt vncnt(0)		1
3464444413018481	65535	1	s_waitcnt vncnt(0)		1
3464444413018481	65535	1	s_endpgm		1
3464444413018481	65535	1	s_endpgm		1
3464444413019601	65535	1	s_waitcnt vncnt(0)		1

continues on next page

Table 1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dis- patch_Id	Instruction	Instruc- tion_Comm	Correlation_Id
3464444413019761	65535	1	s_load_dwor s8, s[4:5], 0x24		1
3464444413019761	65535	1	s_waitcnt vmcnt(0)		1
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_endpgm		1
3464444413020881	65535	1	s_waitcnt lgkmcnt(0)		1
3464444413020881	65535	1	v_adde_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

continues on next page

Table 1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dis- patch_Id	Instruction	Instruc- tion_Comm	Correlation_Id
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
3464444413022161	65535	1	s_load_dword 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_dword 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

continues on next page

Table 1 – continued from previous page

Sample_Timestamp	Exec_Mask	Dis- patch_Id	Instruction	Instruc- tion_Comment	Correlation_Id
3464444413027761	65535	1	s_waitcnt vmcnt(0)		1
3464444413025361	65535	1	s_endpgm		1
3464444413027601	65535	1	s_waitcnt vmcnt(0)		1
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_dword 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 2: PC sampling host trap with debug symbols

Sample_Timestamp	Exec_Mask	Dis-patch_Id	Instruction	Instruction_Comment	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/M
54155306462755	65535	1	s_endpgm	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306462755	65535	1	s_endpgm	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306462955	65535	1	s_endpgm	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463035	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463235	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463315	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463515	65535	1	s_endpgm	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463755	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306463875	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
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/M
54155306464155	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306464275	65535	1	s_endpgm	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
54155306464395	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
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/M
54155306464595	65535	1	s_waitcnt vmcnt(0)	/opt/rocm- 6.4.0/share/h	1 atrixTranspose/M
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.

## 11.2 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)

```

    },
    "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*.

### 11.3 Host-trap PC sampling and arbitrary sampling skid

Host-trap PC sampling is a software-based technique that utilizes a background kernel thread to periodically interrupt running waves to capture the program counter (PC). This method is effective for gathering performance data without requiring specialized hardware to snapshot the waves. However, this method has certain limitations due to the potential delay between receiving and processing the interrupt by the wave to capture the PC. This delay can lead to a sampling skid, where the PC samples might be attributed to the instructions that are up to two instructions away from the actual

source of latency. This results in a non-precise intra-kernel sampling method.

When analyzing an application profile generated by host-trap PC sampling, it is important to consider not only the costliest reported instruction but also the instructions immediately preceding or following it. If the costly instruction is near a branch instruction, it is important to consider the instruction targeted by the branch and the one immediately following it as well.

To address the limitations of host-trap sampling, the hardware-based stochastic PC sampling method has been developed. This method provides precise intra-kernel sampling with zero sampling skid, offering more accurate performance insights.

It is important to note that the skid issue inherent in host-trap PC sampling is not likely to be resolved in its current form. Therefore, to achieve more precise performance profiling, it is recommended to adopt stochastic PC sampling starting with the gfx942 architecture.

#### **Note**

Host-trap PC sampling is supported on AMD Instinct MI200, MI300, MI325, MI350, and MI355.

## 11.4 Hardware-based (stochastic) PC sampling method

The `ROCProfiler_PC_Sampling_Method_Stochastic` method has been introduced for the gfx942 architecture. It employs a specific hardware for probing waves actively running on the GPU. Besides the information already provided with `ROCProfiler_PC_Sampling_Method_Host_Trap` useful for determining hotspots within the kernel, this method delivers additional information, which helps to determine whether a sampled wave issued an instruction represented with the specified PC. If not, this method provides the reason for not issuing the instruction (stall reason). Such information is particularly useful for understanding stalls during kernel execution.

To use this method on gfx942, it is recommended to list available PC sampling configurations to verify if the latest ROCm stack is installed on the system using:

```
rocprofv3 -L
```

An output similar to the following indicates that the `ROCProfiler_PC_Sampling_Method_Stochastic` method is available:

```
GPU: 1
NAME: gfx942
configs:
  Method          :stochastic
  Unit            :cycle
  Min_Interval    :256
  Max_Interval    :2147483648
  Flags           :interval pow2
```

#### **Note**

On gfx942, `ROCProfiler_PC_Sampling_Method_Stochastic` requires intervals to be specified in cycles with values as powers of 2.

To profile an application with `ROCProfiler_PC_Sampling_Method_Stochastic` PC sampling enabled on gfx942, use:

```
rocprofv3 --pc-sampling-beta-enabled --pc-sampling-method stochastic --pc-sampling-unit_
↪cycles --pc-sampling-interval 1048576 --output-format csv, json -- <application_path>
```

The preceding command serializes samples in both CSV and JSON output formats in the `pc_sampling_stochastic.csv` and `out_results.json` files, respectively.

On comparing the `pc_sampling_stochastic.csv` to `pc_sampling_host_trap.csv`, you can notice that the `ROCProfiler_PC_SAMPLING_METHOD_STOCHASTIC` method generates the following additional fields:

- **Wave\_Issued\_Instruction:** Indicates whether the wave issued an instruction represented with the specified PC. Value = 1 for yes and 0 for no.
- **Instruction\_Type:** If the value of `Wave_Issued_Instruction` is 1, this field indicates the type of the issued instruction. Otherwise, this field remains irrelevant.
- **Stall\_Reason:** If the value of `Wave_Issued_Instruction` is 0, this field indicates the reason for not issuing the instruction (stall reason). Otherwise, this field remains irrelevant.
- **Wave\_Count:** Total number of waves actively running on a compute unit when the sample is generated.

Table 3: PC sampling stochastic with debug symbols

Sam- ple_Timestar	Exec_	Dis- patch	In- struc- tion	In- struc- tion_C	Correla- tion_Id	Wave_	Instruc- tion_Type	Stall_Reason	Wave_Count
390705261841	18446	24	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	24	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4
390705261924	18446	29	v_max v1, v2, v0	/home/ sdk- interna operati ops.cpj	29	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6
390705694732	18446	53	v_mad v[0:1], s[2:3], v0, s2, v[2:3]	/home/ sdk- interna operati ops.cpj	53	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6
390705694744	18446	54	v_lshl v[0:1], s[4:5], 0, v[0:1]	/home/ sdk- interna operati ops.cpj	54	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4
390705694769	18446	56	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	56	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6
390705694772	18446	56	s_wait lgkm- cnt(0)	/usr/inc	56	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5

continues on next page

Table 3 – continued from previous page

Sam- ple_Timestar	Exec_	Dis- patch_	In- struc- tion	In- struc- tion_C	Correla- tion_Id	Wave_	Instruc- tion_Type	Stall_Reason	Wave_Count	
390705694810	18446	58	v_cmp vcc, s2, v1	/home/ sdk- interna operati ops.cpj	58	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	INSTRUCTION
390705694820	18446	59	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	59	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	INSTRUCTION
390705694840	18446	60	s_and_ s5, s4, 0xffff	/usr/inc	60	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION
390705694856	18446	61	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	61	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION
390706112944	18446	65	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	65	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	INSTRUCTION
390706112965	18446	66	global_ v[0:1], v2, off	/home/ sdk- interna operati ops.cpj	66	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	INSTRUCTION
390706112966	18446	66	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	66	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	INSTRUCTION
390706112966	18446	66	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	66	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	INSTRUCTION
390706112967	18446	66	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	66	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	INSTRUCTION
390706112971	18446	66	s_load s[4:7], s[0:1], 0x0	/usr/inc	66	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION

continues on next page

Table 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	
390706112984	18446	67	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	67	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	<del>INSTRUCTION</del>
390706112988	18446	67	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	67	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>INSTRUCTION</del>
390706113000	18446	68	v_add v0, s3, v0	/home/ sdk- interna operati ops.cpj	68	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	<del>INSTRUCTION</del>
390706113004	18446	68	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	68	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>INSTRUCTION</del>
390706113053	18446	69	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	69	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	<del>INSTRUCTION</del>
390706113059	18446	69	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	69	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>INSTRUCTION</del>
390706113080	18446	70	s_load s[4:7], s[0:1], 0x0	/usr/inc	70	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>INSTRUCTION</del>
390706113097	18446	71	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	71	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>INSTRUCTION</del>
390706113101	18446	71	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	71	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>INSTRUCTION</del>

continues on next page

Table 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	
390706113111'	18446'	72	v_sub_	/home/	72	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	ON INSTRUCTION
			v4,	sdk-						
			v2,	interna						
			v3	operati						
				ops.cpj						
390706113115'	18446'	72	s_wait	/home/	72	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113134'	18446'	73	s_wait	/home/	73	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113147'	18446'	74	s_wait	/home/	74	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	ON INSTRUCTION
			lgkm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113149'	18446'	74	s_wait	/home/	74	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113153'	18446'	74	s_wait	/home/	74	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113179'	18446'	76	s_wait	/usr/inc	76	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	2	ON INSTRUCTION
			lgkm-							
			cnt(0)							
390706113184'	18446'	76	s_wait	/home/	76	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113184'	18446'	76	s_wait	/home/	76	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						
390706113206'	18446'	77	s_wait	/home/	77	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	10	ON INSTRUCTION
			vm-	sdk-						
			cnt(0)	interna						
				operati						
				ops.cpj						

continues on next page

Table 3 – continued from previous page

Sam- ple_Timestar	Exec_	Dis- patch_	In- struc- tion	In- struc- tion_C	Correla- tion_Id	Wave_	Instruc- tion_Type	Stall_Reason	Wave_Count	
390706113209	18446	77	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	77	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	11	<del>INSTRUC</del>
390706113220	18446	78	s_load s4, s[0:1], 0x2c	/usr/inc	78	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	1	<del>INSTRUC</del>
390706113221	18446	78	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	78	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>INSTRUC</del>
390706113227	18446	78	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	78	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	<del>INSTRUC</del>
390706113234	18446	79	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	79	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>INSTRUC</del>
390706113235	18446	79	s_load s[0:1], s[0:1], 0x10	/usr/inc	79	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>INSTRUC</del>
390706113236	18446	79	s_and_ s[2:3], vcc	/home/ sdk- interna operati ops.cpj	79	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>INSTRUC</del>
390706113250	18446	80	s_wait lgkm- cnt(0)	/usr/inc	80	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	1	<del>INSTRUC</del>
390706113253	18446	80	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	80	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	<del>INSTRUC</del>
390706113255	18446	80	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	80	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>INSTRUC</del>

continues on next page

Table 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	
390706113257	18446	80	v_add_v2, 1.0, v2	/home/sdk-interna-operati-ops.cpj	80	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	4	INSTRUCTION
390706113270	18446	81	s_waitlgkm-cnt(0)	/usr/inc	81	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	INSTRUCTION
390706113278	18446	81	s_loads2, s[0:1], 0x18	/usr/inc	81	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	8	INSTRUCTION
390706113292	18446	82	s_waitvm-cnt(0)	/home/sdk-interna-operati-ops.cpj	82	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	INSTRUCTION
390706113301	18446	83	s_waitlgkm-cnt(0)	/usr/inc	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1	INSTRUCTION
390706113301	18446	83	s_and_s[2:3], vcc	/home/sdk-interna-operati-ops.cpj	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	INSTRUCTION
390706113303	18446	83	s_and_s[2:3], vcc	/home/sdk-interna-operati-ops.cpj	83	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	INSTRUCTION
390706113305	18446	83	v_lshlv[0:1], 2, v[0:1]	/home/sdk-interna-operati-ops.cpj	83	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	INSTRUCTION
390706113317	18446	84	v_div_v3, v3, v6, v7	/home/sdk-interna-operati-ops.cpj	84	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5	INSTRUCTION
390706113318	18446	84	s_waitvm-cnt(0)	/home/sdk-interna-operati-ops.cpj	84	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6	INSTRUCTION

continues on next page

Table 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	
390706113336	18446	85	global_v2, v[2:3], off	/home/sdk-intern-operati-ops.cpj	85	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	11	ON INSTRUCTION
390706113351	18446	86	s_mul_s2, s2, s5	/home/sdk-intern-operati-ops.cpj	86	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	3	ON INSTRUCTION
390706113352	18446	86	s_wait-vm-cnt(0)	/home/sdk-intern-operati-ops.cpj	86	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5	ON INSTRUCTION
390706113369	18446	87	s_wait-vm-cnt(1)	/home/sdk-intern-operati-ops.cpj	87	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	6	ON INSTRUCTION
390706113373	18446	87	s_wait-vm-cnt(1)	/home/sdk-intern-operati-ops.cpj	87	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5	ON INSTRUCTION
390706113387	18446	88	s_wait-igkm-cnt(0)	/usr/inc	88	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	1	ON INSTRUCTION
390706113390	18446	88	s_wait-vm-cnt(0)	/home/sdk-intern-operati-ops.cpj	88	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5	ON INSTRUCTION
390706113408	18446	89	s_wait-vm-cnt(0)	/home/sdk-intern-operati-ops.cpj	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	5	ON INSTRUCTION
390706113409	18446	89	v_add_v2, v2, v3	/home/sdk-intern-operati-ops.cpj	89	1	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	10	ON INSTRUCTION
390706113411	18446	89	s_wait-vm-cnt(0)	/home/sdk-intern-operati-ops.cpj	89	0	ROCPRO-FILER_PC_SA	ROCPRO-FILER_PC_SA	9	ON INSTRUCTION

continues on next page

Table 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	
390706113411	18446	89	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	89	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	<del>UNINSPECTION</del>
390706113412	18446	89	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	89	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	<del>UNINSPECTION</del>
390706113415	18446	89	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	89	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	9	<del>UNINSPECTION</del>
390706113424	18446	90	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	90	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>UNINSPECTION</del>
390706113443	18446	91	v_add_ v2, -1.0, v2	/home/ sdk- interna operati ops.cpj	91	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	2	<del>UNINSPECTION</del>
390706113444	18446	91	v_add_ v3, -1.0, v3	/home/ sdk- interna operati ops.cpj	91	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>UNINSPECTION</del>
390706113459	18446	92	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	92	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>UNINSPECTION</del>
390706113459	18446	92	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	92	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>UNINSPECTION</del>
390706113462	18446	92	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	92	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	<del>UNINSPECTION</del>

continues on next page

Table 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	
390706113480	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	15	<del>UNINSPECTION</del>
390706113484	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	7	<del>UNINSPECTION</del>
390706113496	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>UNINSPECTION</del>
390706113500	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	9	<del>UNINSPECTION</del>
390706113506	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	14	<del>UNINSPECTION</del>
390706113506	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	<del>UNINSPECTION</del>
390706113508	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	17	<del>UNINSPECTION</del>
390706113522	184467	93	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	93	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	15	<del>UNINSPECTION</del>
390706113561	184467	94	s_wait lgkm- cnt(0)	/home/ sdk- interna operati ops.cpj	94	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	3	<del>UNINSPECTION</del>

continues on next page

Table 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	
390706113573	184465	95	s_wait	/home/ vm- cnt(1)	95	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	UNINSPECTION
390706526501	184465	112	s_wait	/home/ vm- cnt(0)	112	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	UNINSPECTION
390706526582	184465	117	s_wait	/home/ vm- cnt(0)	117	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	8	UNINSPECTION
390706526594	184465	118	s_wait	/home/ vm- cnt(0)	118	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	UNINSPECTION
390706526629	184465	120	s_wait	/home/ lgkm- cnt(0)	120	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	UNINSPECTION
390706526629	184465	120	s_wait	/home/ vm- cnt(0)	120	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	UNINSPECTION
390706526633	184465	120	v_mul	/home/ v7, v3, v6	120	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	UNINSPECTION
390706526634	184465	120	s_wait	/home/ vm- cnt(0)	120	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	UNINSPECTION
390706526665	184465	122	s_wait	/home/ vm- cnt(0)	122	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	4	UNINSPECTION

continues on next page

Table 3 – continued from previous page

Sam- ple_Timestar	Exec_	Dis- patch_	In- struc- tion	In- struc- tion_C	Correla- tion_Id	Wave_	Instruc- tion_Type	Stall_Reason	Wave_Count	
390706526677	18446	123	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	123	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	6	INSTRUCTION
390706526695	18446	124	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	124	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION
390706526809	18446	126	v_and_ v2, 0x7fff v2	/opt/ro 6.4.0/li	126	1	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION
390706526810	18446	126	s_wait vm- cnt(0)	/home/ sdk- interna operati ops.cpj	126	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION
390706526821	18446	127	s_wait vm- cnt(1)	/home/ sdk- interna operati ops.cpj	127	0	ROCPRO- FILER_PC_SA	ROCPRO- FILER_PC_SA	5	INSTRUCTION

Similarly, ROCPROFILER\_PC\_SAMPLING\_METHOD\_STOCHASTIC method delivers additional information to every sample in the JSON output.

Here is a `out_results.json` file sample:

```
{
  "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": {
```

(continues on next page)

(continued from previous page)

```

    "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,
    "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. For example, `arb_state_issue_` fields indicate the type of instructions issued by the arbiter at the time of sampling. On the other hand, `arb_state_stall_` fields indicate the type of instructions stalled at the time of sampling. This information is useful for understanding how many instructions per cycle (IPC) are issued.

**i Note**

The stochastic PC sampling is supported on AMD Instinct MI300, MI325, MI350, and MI355.

## 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.

### 12.1 Prerequisites

- aqprofile:
  - 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`

## 12.2 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	String	Any		Profiles kernel names matching the regex
kernel-exclude-regex	String	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	Boolean		False	If set to "True", turns on serialization for untraced kernels
att-perfcounter-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-perfcounter-activity	String	SQ-only		Available only in gfx9. Specifies the list of SQ counters. To list all counters, use "rocprofv3 -list-avail".
att-gpu-index	Integer (List)	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.
att-consecutive-kernels	Integer	>=0		Comma-separated list of integers. If enabled, only the GPU indexes in the list will be profiled by thread trace.
				Starting at the targeted kernel, enables thread trace for the next N kernel dispatches, sharing a single ATT file, stats.csv and UI dir. See -kernel-include-regex and -kernel-iteration-range. If multiple targeted kernels overlap, the count for N next dispatches starts again from 0. Recommended use with -att-gpu-index due to thread trace being enabled for all GPUs.

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>
```

## 12.3 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"
    }
  ]
}
```

## 12.4 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.

Typically, each kernel profile has its own ATT file output. To compile multiple kernel profiles into a single output file, use the `att-consecutive-kernels` parameter. When using this parameter, the rocprofv3 tool begins profiling kernels after encountering a targeted kernel. The tool then continues profiling subsequent kernels until a total of `n` kernels are profiled including the initial targeted kernel where `n` is the non-negative integer passed to `att-consecutive-kernels`. Note that the subsequent kernels encountered after the initial targeted kernel do not themselves have to be targeted. If the subsequent kernels are targeted kernels, the profiler will then profile another `n - 1` kernels after encountering this new targeted kernel, so it is possible for a generated ATT file to have more than `n` kernels profiled. All the profiled kernels are then compiled into a single ATT file. If a new targeted kernel is encountered after the rocprofv3 tool has finished profiling a batch of kernels, the profiler will restart profiling when encountering this new targeted kernel and create another ATT file with multiple kernels.

## 12.5 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.

### 12.5.1 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	kernel.py:391
11	5896	s_load_dwordx2 s[38:39], s[0:1], 0x28	48	192	0	0	kernel.py:391
11	5904	s_ashr_i32 s3, s2, 31	48	260	0	0	kernel.py:395
11	5908	s_add_i32 s7, s2, s3	48	196	0	0	kernel.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.

## 12.6 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.

## 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.

### 13.1 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.

### 13.2 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)

```

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.

### 13.3 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.

## 13.4 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 roc_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;
}
}

```



## 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.

### 14.1 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")));
}
```

### 14.2 Internal implementation of API function

```
namespace impl
{
int
foo(int val)
{
// real implementation
return (2 * val);
}
}
```

### 14.3 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

```

## 14.4 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);
    }
}

```

## 14.5 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](#).

## IMPLEMENTING PROCESS ATTACHMENT TOOLS

### 15.1 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 can use either the provided python or exported C functions.

### 15.2 Python Functions

The python file `rocprofv3-attach.py` defines a main function that can be used for attachment:

```
def main(  
    pid=os.environ.get("ROCPROF_ATTACH_PID", None),  
    attach_library=os.environ.get(  
        "ROCPROF_ATTACH_TOOL_LIBRARY", ROCPROF_ATTACH_TOOL_LIBRARY  
    ),  
    duration=os.environ.get("ROCPROF_ATTACH_DURATION", None),  
)
```

#### Function Details

The main function performs the entire attachment process, including attaching and detaching, and provides the ability to use custom tools. It also has simple control flow intended for direct calling from python. For more complex control, it is recommended to instead use the explicit attach and detach functions provided by the `librocprofv3-attach.so` binary.

#### Parameters

- **pid: Required - PID of process to attach to**
  - Defaults to environment variable `ROCPROF_ATTACH_PID`
- **attach\_library: Optional - Colon delimited list of tool libraries to use**
  - Defaults to environment variable `ROCPROF_ATTACH_TOOL_LIBRARY`
  - If unspecified, defaults to the absolute path of `librocprofv3-attach.so`
- **duration: Optional - Length of time in milliseconds to profile for**
  - Defaults to environment variable `ROCPROF_ATTACH_DURATION`
  - If unspecified, attachment will run until Enter is pressed or SIGINT (Ctrl+C) is received

## 15.3 C Functions

The C library `librocprofv3-attach.so` defines an attach and detach function that can be used for attachment:

```
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
  - Custom tool libraries can be specified in a colon delimited list with the environment variable `ROC_PROF_ATTACH_TOOL_LIBRARY`
- **detach(): Entry point for detaching from the target process**
  - Cleans up attachment resources and terminates profiling

## 15.4 Function Call Sequence

### 15.4.1 Initial Attachment Sequence

The initial attachment process roughly follows this sequence:

1. `attach(pid)` ← Your tool calls this
2. `ptrace` calls `rocprofiler_register_attach(env_buffer)`
3. `tool_library::rocprofiler_configure(...)`
4. `tool_library::rocprofiler_configure_attach(...)`
5. `tool_library::tool_init(...)`
6. `tool_library::tool_attach(...)`
7. [Profiling and data collection...]
8. `detach()` ← Your tool calls this
9. `ptrace` calls `rocprofiler_register_detach()`
10. `tool_library::tool_detach(...)`
11. [Program ends]
12. `tool_library::tool_fini(...)`

### 15.4.2 Reattachment Sequence

For reattachment to a previously attached process:

1. `attach(pid)` ← Your tool calls this again
2. `ptrace` calls `rocprofiler_register_attach(env_buffer)`

3. tool\_library::tool\_attach(...)
4. [Continued profiling and data collection...]
5. detach() ← Your tool calls this
6. ptrace calls rocprofiler\_register\_detach()
7. tool\_library::tool\_detach(...)

## 15.5 Environment Variable Configuration

The target process must have ROCP\_TOOL\_ATTACH=1 set, or be using a version of rocprofiler-register configured with the CMake flag ROCPROFILER\_REGISTER\_BUILD\_DEFAULT\_ATTACHMENT=ON

### 15.5.1 Required Variables

```
export ROCP_TOOL_ATTACH=1
OR
cmake /path/to/rocprofiler-register -DROCPROFILER_REGISTER_BUILD_DEFAULT_ATTACHMENT=ON
```

### 15.5.2 Tool Library Configuration

The attachment system can use any tool library. librocprofiler-sdk-tool.so is used when the environment variable is not set.

```
// Attachment libraries to be used
setenv("ROC_PROF_ATTACH_TOOL_LIBRARY", "example-tool-1.so:example-tool-2.so", 1);
```

## 15.6 Using the Attachment Functions

Here's how to use these functions in your own attachment tool:

### 15.6.1 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("librocprofv3-attach.so", RTLD_NOW);
        if (!attach_lib_handle) {
            std::cerr << "Failed to load librocprofv3-attach: " << dlerror() << "\n";
        }
    }
};
```

(continues on next page)

(continued from previous page)

```

        return false;
    }

    // Get the attachment function pointers
    attach_func = (void*)(uint32_t)dlsym(attach_lib_handle, "attach");
    detach_func = (void*)(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..." <<
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);
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
};

```

## 15.6.2 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;

    // For this example, the tool library "librocprofiler-sdk-tool.so" is used by
    // default because ROCPROF_ATTACH_TOOL_LIBRARY is not set. These environment
    // variables are used to communicate profiling options to rocprofiler-sdk-tool.
    setenv("ROCPROF_HIP_RUNTIME_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;
}

```



## 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.

### 16.1 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.

### 16.2 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.

## 16.3 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);
}
```

## 16.4 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. ...
}

```

## 16.5 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"};
    }
}
}

```

## 16.6 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.



## 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.

### 17.1 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. ...
}

```

## 17.2 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)

```

↪user_data,
rocprofiler_user_data_t*
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.*

## 17.3 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

```

## 17.4 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](#).



## 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](#).

### 18.1 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 (note: this field is not set.
↪externally)
ROCPROFILER_DIMENSION_SHADER_ARRAY,  ///< Number of shader arrays
ROCPROFILER_DIMENSION_WGP,           ///< Number of workgroup processors
ROCPROFILER_DIMENSION_INSTANCE,      ///< From unspecified hardware register
```

## 18.2 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:

### 18.2.1 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");
```

## 18.3 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.

## 18.4 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`.

## 18.5 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)`.

### 18.5.1 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 << "},"";
}
}

```

## 18.6 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.

## 18.7 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`.

### 18.7.1 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

## 18.7.2 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

## 18.8 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_NUM
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.

## 18.9 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.



## 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.

## 19.1 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](#).

### 19.1.1 `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](#).

## 19.2 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)

```
        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`.

## 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](#).

## 20.1 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](#).

### 20.1.1 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,
    ↪OBJECT,
    ROCPROFILER_CALLBACK_TRACING_CODE_
    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 `ROC_PROFILER_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.

### 20.1.2 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)
{
    ROC_PROFILER_CALL(
        rocprofiler_configure_device_thread_trace_service(
            ctx,
            agent_id,
            params.data(),
            params.size(),
            shader_data_callback,
            nullptr),
        "thread trace service configure");
}
```

### 20.1.3 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({ROC_PROFILER_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 ROC_PROFILER_THREAD_TRACE_CONTROL_START_AND_STOP;

    return ROC_PROFILER_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);

```

## 20.2 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.

### 20.2.1 Trace Decoder setup

To decode the raw thread trace data, create and initialize a Trace Decoder:

```

rocprofiler_thread_trace_decoder_id_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");

```

## 20.2.2 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");
}

```

## 20.3 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);
}

```

### 20.3.1 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
    }
}

```

### 20.3.2 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](#)



## ROCPROFILER-SDK API LIBRARY

This ROCprofiler-SDK API topic covers:

- *Modules*
- *Global Data structures, topics, files*

### 21.1 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*

#### 21.1.1 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\_visibility\_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.

## 21.1.2 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*

### 21.1.3 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&lt;&lt; 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 parameter 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 parameter 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.\_\_unnamed2\_\_**

### 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]**

## 21.1.4 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, rocpro-
                                                                filer_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) &#8212; 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_t
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.\_\_unnamed4\_\_**

### **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.\_\_unnamed6\_\_**

### **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.\_\_unnamed6\_\_.\_\_unnamed8\_\_**

**rocprofiler\_callback\_tracing\_code\_object\_load\_data\_t.\_\_unnamed6\_\_.\_\_unnamed10\_\_**

## 21.1.5 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.

## 21.1.6 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*

## 21.1.7 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 available 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.

### 21.1.8 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 successfully 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 successfully 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*

## 21.1.9 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, rocpro-
filer_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*

## 21.1.10 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_request_kinds_t *kinds,
unsigned long kinds_count,
rocprofiler_external_correlation_id_request_callback_t 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) &#8212; 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\_t* 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 parameter 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 &#8212; 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*

### 21.1.11 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 parameter 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)

## 21.1.12 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. `ROC_PROFILER_LIBRARY | ROC_PROFILER_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

`ROC_PROFILER_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

- `ROC_PROFILER_STATUS_SUCCESS` – Successful thread creation
- `ROC_PROFILER_STATUS_ERROR_CONFIGURATION_LOCKED` – Thread creation is no longer available post-initialization
- `ROC_PROFILER_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 deliver 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

- `ROC_PROFILER_STATUS_SUCCESS` – Successful assignment of the delivery thread for the given buffer
- `ROC_PROFILER_STATUS_ERROR_CONFIGURATION_LOCKED` – Thread assignment is no longer available post-initialization
- `ROC_PROFILER_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*

### 21.1.13 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 its 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\*

### 21.1.14 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 requested 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 contain 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:
  - PC sampling is requested from a process that runs within the ROCgdb.
  - 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_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.

### 21.1.15 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. gfx9 only.

enumerator **ROCProfiler\_THREAD\_TRACE\_PARAMETER\_SERIALIZE\_ALL**

Serializes also kernels not under thread trace.

enumerator **ROCProfiler\_THREAD\_TRACE\_PARAMETER\_PERFCOUNTER\_EXCLUDE\_MASK**

Bitmask of which compute units to exclude from perfcounters. gfx9 only.

enumerator **ROCProfiler\_THREAD\_TRACE\_PARAMETER\_NO\_DETAIL**

Dont collect instruction timing, only shader-wide information.

enumerator `ROC_PROFILER_THREAD_TRACE_PARAMETER_LAST`

enum `rocprofiler_thread_trace_control_flags_t`

*Values:*

enumerator `ROC_PROFILER_THREAD_TRACE_CONTROL_NONE`

enumerator `ROC_PROFILER_THREAD_TRACE_CONTROL_START_AND_STOP`

enum `rocprofiler_thread_trace_decoder_info_t`

Describes the type of info received.

*Values:*

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_INFO_NONE`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_INFO_DATA_LOST`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_INFO_STITCH_INCOMPLETE`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_INFO_WAVE_INCOMPLETE`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_INFO_LAST`

enum `rocprofiler_thread_trace_decoder_wstate_type_t`

Wave state type.

*Values:*

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_WSTATE_EMPTY`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_WSTATE_IDLE`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_WSTATE_EXEC`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_WSTATE_WAIT`

enumerator `ROC_PROFILER_THREAD_TRACE_DECODER_WSTATE_STALL`

enumerator `ROC_PROFILER_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\_shaderdata\_flags\_t**

Bitmask of additional information for `shaderdata_t` Added in rocprof-trace-decoder 0.1.3.

*Values:*

enumerator **ROCProfiler\_THREAD\_TRACE\_DECODER\_SHADERDATA\_FLAGS\_IMM**

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 `uint64_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\_SHADERDATA**

`rocprofiler_thread_trace_decoder_shaderdata_t*`

enumerator **ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_REALTIME**

`rocprofiler_thread_trace_decoder_realtime_t*`

enumerator **ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_RT\_FREQUENCY**

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, rocpro-
filer_thread_trace_parameter_t
*parameters, unsigned long
num_parameters, rocpro-
filer_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\_id\_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\_id\_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\_id\_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\_id\_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\_id\_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\_id\_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\_id\_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 its 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.

struct **rocprofiler\_thread\_trace\_decoder\_realtime\_t**

*#include <rocprofiler-sdk/experimental/thread-trace/trace\_decoder\_types.h>* Matches the reference (realtime) clock with the shader clock Added in rocprof-trace-decoder 0.1.3. Requires aqlprofile for rocm 7.1+.  $clock\_in\_seconds = realtime\_clock / ROCPROFILER\_THREAD\_TRACE\_DECODER\_RECORD\_RT\_FREQUENCY$   $gfx\_frequency = \frac{\delta(shader\_clock)}{\delta(clock\_in\_seconds)}$  For best average, use  $gfx\_frequency[n] = \frac{shader\_clock[n]-shader\_clock[0]}{clock\_in\_seconds[n]-clock\_in\_seconds[0]}$

struct **rocprofiler\_thread\_trace\_decoder\_shaderdata\_t**

*#include <rocprofiler-sdk/experimental/thread-trace/trace\_decoder\_types.h>* Record created by s\_ttracedata and s\_ttracedata\_imm Added in rocprof-trace-decoder 0.1.3.

**rocprofiler\_thread\_trace\_parameter\_t.\_\_unnamed12\_\_**

### Public Members

uint64\_t value

struct rocprofiler\_thread\_trace\_parameter\_t

**rocprofiler\_thread\_trace\_parameter\_t.\_\_unnamed12\_\_.\_\_unnamed14\_\_**

## 21.1.16 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* function pointer during its 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
        // 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,
```

(continues on next page)

(continued from previous page)

```

        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
    return &cfg;
}

```

**Parameters**

- **version** – [in] The version of rocprofiler:  $(10000 * \text{major}) + (100 * \text{minor}) + \text{patch}$
- **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> &#8212; at least that is the plan). The handle member is a unique identifier 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 &#8212; 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)`

## 21.2 Global Data structures, topics, files

This ROCprofiler-SDK API topic covers:

- *Global Basic Data Types*
- *Topics*
- *Data Structures*
- *File List*

### 21.2.1 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 **ROC\_PROFILER\_CALLBACK\_TRACING\_HSA\_AMD\_EXT\_API**

 **See also**

rocprofiler\_hsa\_amd\_ext\_api\_id\_t

enumerator **ROC\_PROFILER\_CALLBACK\_TRACING\_HSA\_IMAGE\_EXT\_API**

 **See also**

rocprofiler\_hsa\_image\_ext\_api\_id\_t

enumerator **ROC\_PROFILER\_CALLBACK\_TRACING\_HSA\_FINALIZE\_EXT\_API**

 **See also**

rocprofiler\_hsa\_finalize\_ext\_api\_id\_t

enumerator **ROC\_PROFILER\_CALLBACK\_TRACING\_HIP\_RUNTIME\_API**

 **See also**

rocprofiler\_hip\_runtime\_api\_id\_t

enumerator **ROC\_PROFILER\_CALLBACK\_TRACING\_HIP\_COMPILER\_API**

 **See also**

rocprofiler\_hip\_compiler\_api\_id\_t

enumerator **ROC\_PROFILER\_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 reclamation 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 interval 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 particular 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 being 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**

compatibility

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.\_\_unnamed339\_\_**

## 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.\_\_unnamed339\_\_.\_\_unnamed344\_\_**

**21.2.2 Topics**

**21.2.3 Data Structures**

**21.2.4 File List**



## 22.1 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*

## 22.2 Modules

The ROCprofiler-SDK-ROCTx API is organized into the following modules based on functionality:

- *Markers Information*
- *Ranges Information*
- *Profiler Control Information*
- *Naming Information*

### 22.2.1 Markers Information

void **roctxMarkA**(const char \*message)

Mark an event.

#### Parameters

**message** – [in] The message associated with the event.

### 22.2.2 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

### 22.2.3 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.

### 22.2.4 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

## 22.3 Global Data structures, topics, files

This ROCprofiler-SDK-ROCTx API topic covers:

- *Global Basic Data Types*
- *Topics*
- *File List*

### 22.3.1 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

### 22.3.2 Topics

### 22.3.3 File List



## 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.



## 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 1: Comparison of ROCprofiler Command-Line Tool's options

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Basic tracing options	HIP Trace	<i>-hip-trace</i>	<i>-hip-api,</i> <i>-hip-trace</i>	<i>-hip-trace</i>	No change	rocprof and rocprofv2 <i>-hip-trace</i> options include kernel dispatches and memory copy activities, which is not the case in rocprofv3
Basic tracing options	HSA Trace	<i>-hsa-trace</i>	<i>-hsa-trace</i>	<i>-hsa-trace</i>	No change	rocprof and rocprofv2 <i>-hsa-trace</i> options include kernel dispatches and memory copy activities, which is not the case in rocprofv3

continues on next page

---

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Basic tracing options	Scratch Memory Trace	<i>Not Available</i>	<i>Not Available</i>	<i>-scratch-memory-trace</i>	New option to trace scratch memory operations	
Basic tracing options	Marker Trace (ROCTX)	<i>-roctx-trace</i>	<i>-roctx-trace</i>	<i>-marker-trace</i>	Improved ROCTX library with more features	
Basic tracing options	Memory Copy Trace	Part of HIP and HSA Traces	Part of HIP and HSA Traces	<i>-memory-copy-trace</i>	Provides granularity for memory move operations	
Basic tracing options	Memory allocation Trace	<i>Not Available</i>	<i>Not Available</i>	<i>-memory-allocation-trace</i>	New option for collecting Memory Allocation Traces. Displays starting address, allocation size, and agent where allocation occurred.	
Basic tracing options	Kernel Trace	<i>-kernel-trace</i>	<i>-kernel-trace</i>	<i>-kernel-trace</i>	Performance improvement.	
Granular tracing options	HIP runtime trace	Part of <i>-hip-trace</i> option	Part of <i>-hip-trace</i> option	<i>-hip-runtime-trace</i>	For collecting HIP Runtime API Traces, e.g. public HIP API functions starting with 'hip' (i.e. hipSetDevice).	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Granular tracing options	HIP compiler trace	<i>Not Available</i>	<i>Not Available</i>	<i>-hip-compiler-trace</i>	For collecting HIP Compiler generated code Traces, e.g. HIP API functions starting with ‘ <i>__hip</i> ’ (i.e. <i>__hipRegisterFatBinary</i> ).	
Granular tracing options	HSA core API trace	Part of <i>-hsa-trace</i> option	Part of <i>-hsa-trace</i> option	<i>-hsa-core-trace</i>	New option for collecting only HSA API Traces (core API), e.g. HSA functions prefixed with only <i>hsa_</i> (i.e. <i>hsa_init</i> )	
Granular tracing options	HSA AMD trace	Part of <i>-hsa-trace</i> option	Part of <i>-hsa-trace</i> option	<i>-hsa-amd-trace</i>	For collecting HSA API Traces (AMD-extension API), e.g. HSA function prefixed with <i>hsa_amd_</i> (i.e. <i>hsa_amd_coher</i>	
Granular tracing options	HSA Image Extension trace	Part of <i>-hsa-trace</i> option	Part of <i>-hsa-trace</i> option	<i>-hsa-image-trace</i>	New option for collecting HSA API Traces (Image-extension API), e.g. HSA functions prefixed with only <i>hsa_ext_image_</i> (i.e. <i>hsa_ext_image_</i>	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Granular tracing options	HSA Finalizer trace	Part of <i>-hsa-trace</i> option	Part of <i>-hsa-trace</i> option	<i>-hsa-finalizer-trace</i>	New option for collecting HSA API Traces (Finalizer-extension API), e.g. HSA functions prefixed with only <i>hsa_ext_progra</i> (i.e. <i>hsa_ext_progra</i> )	
Advanced tracing options	Kokkos trace	<i>Not Available</i>	<i>Not Available</i>	<i>-kokkos-trace</i>	New option to enable built-in Kokkos Tools support (implies <i>-marker-trace</i> and <i>-kernel-rename</i> )	
Advanced tracing options	RCCL trace	<i>Not Available</i>	<i>Not Available</i>	<i>-rccl-trace</i>	For collecting RCCL (ROCm Communication Collectives Library. Also pronounced as 'Rickle' ) Traces	
Advanced tracing options	Scratch memory trace	<i>Not Available</i>	<i>Not Available</i>	<i>-scratch-memory-trace</i>	Collecting scratch memory event traces.	
Advanced tracing options	rocDecode trace	<i>Not Available</i>	<i>Not Available</i>	<i>-rocdecode-trace</i>	Tracing rocDecode library.	
Advanced tracing options	rocJPEG trace	<i>Not Available</i>	<i>Not Available</i>	<i>-rocjpeg-trace</i>	Tracing rocJPEG library.	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Aggregate tracing options	Sys Trace	<code>-sys-trace</code> [hip-trace hsa-trace roctx-trace kernel-trace]	<code>-sys-trace</code> [hip-trace hsa-trace roctx-trace kernel-trace]	<code>`-s, -sys-trace`</code> [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	<i>Not available</i>	<i>Not available</i>	<code>`-r, -runtime-trace`</code> [hip-runtime-trace scratch-trace memory-copy-trace roctx-trace kernel-trace]	New option to aggregate trace operations	
Kernel naming options	Kernel Name Mangling	<i>Not Available</i>	<i>Not Available</i>	<code>-M, -mangled-kernels</code>	New option for mangled kernel names	
Kernel naming options	Kernel Name Truncation	<code>-basenames</code> <on off>	<code>-basenames</code>	<code>-T, -truncate-kernels</code>	New option for truncating the demangled kernel names	
Kernel naming options	Kernel Re-name	<code>-roctx-rename</code>	<i>Not available</i>	<code>-kernel-rename</code>	New option to use region names defined by roctxRangePush/roctxRangePop regions to rename the kernels	
Post-processing tracing options	Statistics	<code>-stats</code>	<i>Not Available</i>	<code>-stats</code>	Statistics for the collected traces	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Post-processing tracing options	Summary	<i>Not available</i>	<i>Not available</i>	<i>-S, -summary</i>	New option to output a single summary of tracing data after the profiling session	<i>rocprof</i> generated the post-processing step's summary, stats, JSON, and database files with much less information.
Post-processing tracing options	Summary Per Domain	<i>Not available</i>	<i>Not available</i>	<i>-D, -summary-per-domain</i>	New option to output summary for each tracing domain after the profiling session	<i>rocprof -stats</i> option had less number of domains in the summary reports than <i>rocprofv3</i>
Post-processing tracing options	Summary Groups	<i>Not available</i>	<i>Not available</i>	<i>-summary-groups REGULAR_EXPRESSION</i>	New option to output a summary for each set of domains matching the regular expression, e.g. 'KERNEL_DISPATCH' will generate a summary from all the tracing data in the KERNEL_DISPATCH and MEMORY_COPY domains	'COPY'
Summary options	Summary Output File	<i>Not available</i>	<i>Not available</i>	<i>-summary-output-file SUMMARY_OUTPUT</i>	New option to output summary to a file, stdout, or stderr (default: stderr)	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Summary options	Summary Units	<i>Not available</i>	<i>Not available</i>	<i>-u</i> , <i>-summary-units</i>	New option to output summary in desired time units {sec,msec,usec}	
Display options	List available basic and derived metrics and PC sampling configurations	<i>-list-basic</i> , <i>-list-derived</i>	<i>-list-counters</i>	<i>-L</i> , <i>-list-avail</i>	A valid YAML is supported for this option now	
Perfetto-specific options	Perfetto data collection backend	<i>Not available</i>	<i>Not available</i>	<i>-perfetto-backend</i> {in-process,system}	New option for perfetto data collection backend. 'system' mode requires starting traced and perfetto daemons	<i>rocprofv2</i> used only in-process collection for perfetto plugin. However, <i>rocprofv3</i> gives the user the option.
Perfetto-specific options	Perfetto Buffer Size	<i>Not available</i>	Setting env variable <i>rocprofiler_PERFETTO</i> to the desired buffer size	<i>-perfetto-buffer-size</i> {KB}	New option to define size of buffer for perfetto output in KB. default: 1 GB	
Perfetto-specific options	Perfetto Buffer fill Policy	<i>Not available</i>	<i>Not available</i>	<i>-perfetto-buffer-fill-policy</i> {discard,ring_buffer}	New option or handling new records when perfetto has reached the buffer limit	<i>rocprofv2</i> always used <i>TraceConfig_BufferConfig_FillPolicy_RING</i> fill policy.
Perfetto-specific options	Perfetto shared memory size	<i>Not available</i>	<i>Not available</i>	<i>-perfetto-shmem-size-hint</i> KB	New option to define perfetto shared memory size hint in KB. default: 64 KB	

continues on next page

Table 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)	<i>-kernel-include-regex</i> , <i>-kernel-exclude-regex</i> , <i>-kernel-iteration-range</i>	Extensive control over output options using regular expressions	
I/O options	Output Directory	<i>-d</i> <data directory>	<i>-d</i>   <i>-output-directory</i>	<i>-d</i> OUTPUT_DIRECTORY, <i>-output-directory</i> OUTPUT_DIRECTORY	rocprofv3 supports special keys for runtime values, e.g. %pid% gets replaced by the process ID	
I/O options	Output File	<i>-o</i> <output file>	<i>-o</i>   <i>-output-file-name</i>	<i>-o</i> OUTPUT_FILE, <i>-output-file</i> 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	<i>-log-level</i> {fatal,error,warnin	Extensive logging options	
I/O options	Plugins	<i>Not Available</i>	plugin support for different output formats	Replaced by <i>-output-format</i> option	Not needed as rocprofv3 supports multiple output formats	

continues on next page

Table 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 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
I/O options	Command-line Counter Collection	<i>Not Available</i>	<i>Not Available</i>	<i>-pmc</i>	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	<i>-m &lt;metric file&gt;</i>	<i>-m &lt;metric file&gt;</i>	<i>-E &lt;metric file&gt; -pmc &lt;counter&gt;</i>	In rocprofv3, this option has changed to provide a file with custom metrics and collect performance counters from the command line using <i>-pmc</i> option	
Advanced options	Preload	<i>Not Available</i>	<i>Not Available</i>	<i>-preload</i>	Libraries to prepend to LD_PRELOAD (usually for sanitizers)	
Trace Control options	Trace Period	<i>-trace-period</i>	<i>-tp   -trace-period</i>	<i>-P   -collection-period, -collection-unit`</i>	Users can specify multiple configurations, each defined by a triplet in the format <i>start_delay:col. at,</i> with the ability to change the unit of time in the given configurations.	
Trace Control options	Trace start	<i>-trace-start &lt;on off&gt;</i>	<i>Not available</i>	<i>Not available</i>	Not yet in rocprofv3	

continues on next page

Table 1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Trace Control options	Flush Interval	<i>-flush-rate</i>	<i>-flush-interval</i>	<i>Not available</i>	Not applicable for rocprofv3	
Trace Control options	Merge Traces	<i>-merge-traces</i>	<i>Not available</i>	<i>Not available</i>	Not yet in rocprofv3	
PC Sampling options	PC Sampling`	<i>Not available</i>	<i>Not available</i>	<i>-pc-sampling-beta-enabled</i>	Enable pc sampling support; beta version.	
Legacy options	Timestamp On/Off	<i>-timestamp &lt;on off&gt;</i>	<i>Not available</i>	<i>Not available</i>	Not applicable for rocprofv3	
Legacy options	Context wait	<i>-ctx-wait</i>	<i>Not available</i>	<i>Not available</i>	Not applicable for rocprofv3	
Legacy options	Context Limit	<i>-ctx-limit &lt;max number&gt;</i>	<i>Not available</i>	<i>Not available</i>	Not applicable for rocprofv3	
Legacy options	Code Object Tracking	<i>-obj-tracking &lt;on off&gt;</i>	Always ON in rocprofv2	Always ON in rocprofv3		
Legacy options	Heartbeat	<i>-heartbeat &lt;rate sec&gt;</i>	<i>Not available</i>	<i>Not available</i>	Not applicable for rocprofv3	



## 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.



## 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.



**LICENSE**

MIT License

Copyright (C) Advanced Micro Devices, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## INDEX

### P

PhonyNameDueToError::agent\_id (C++ member), 210  
PhonyNameDueToError::hash (C++ member), 266  
PhonyNameDueToError::reserved (C++ member), 205  
PhonyNameDueToError::rocp\_agent (C++ member), 210  
PhonyNameDueToError::target (C++ member), 205  
PhonyNameDueToError::target\_data\_op (C++ member), 205  
PhonyNameDueToError::target\_kernel (C++ member), 205  
PhonyNameDueToError::value (C++ member), 240

### R

rocprofiler\_address\_t (C++ union), 264  
rocprofiler\_address\_t::handle (C++ member), 264  
rocprofiler\_address\_t::ptr (C++ member), 264  
rocprofiler\_address\_t::value (C++ member), 264  
rocprofiler\_agent\_cache\_t (C++ struct), 198  
rocprofiler\_agent\_id\_t (C++ struct), 265  
rocprofiler\_agent\_io\_link\_t (C++ struct), 198  
rocprofiler\_agent\_mem\_bank\_t (C++ struct), 198  
rocprofiler\_agent\_runtime\_visiblity\_t (C++ struct), 198  
rocprofiler\_agent\_t (C++ type), 198  
rocprofiler\_agent\_type\_t (C++ enum), 247  
rocprofiler\_agent\_type\_t::ROCProfiler\_AGENT\_TYPE\_CPU (C++ enumerator), 247  
rocprofiler\_agent\_type\_t::ROCProfiler\_AGENT\_TYPE\_GPU (C++ enumerator), 248  
rocprofiler\_agent\_type\_t::ROCProfiler\_AGENT\_TYPE\_LAST (C++ enumerator), 248  
rocprofiler\_agent\_type\_t::ROCProfiler\_AGENT\_TYPE\_NONE (C++ enumerator), 247  
rocprofiler\_agent\_v0\_t (C++ struct), 199  
rocprofiler\_agent\_version\_t (C++ enum), 197  
rocprofiler\_agent\_version\_t::ROCProfiler\_AGENT\_INFO\_VERSION\_0 (C++ enumerator), 198

rocprofiler\_agent\_version\_t::ROCProfiler\_AGENT\_INFO\_VERSION\_1 (C++ enumerator), 198  
rocprofiler\_agent\_version\_t::ROCProfiler\_AGENT\_INFO\_VERSION\_2 (C++ enumerator), 197  
rocprofiler\_assign\_callback\_thread (C++ function), 224  
rocprofiler\_async\_correlation\_id\_t (C++ struct), 265  
rocprofiler\_at\_intercept\_table\_registration (C++ function), 223  
rocprofiler\_at\_internal\_thread\_create (C++ function), 223  
rocprofiler\_available\_counters\_cb\_t (C++ type), 212  
rocprofiler\_available\_pc\_sampling\_configurations\_cb\_t (C++ type), 228  
rocprofiler\_buffer\_category\_t (C++ enum), 247  
rocprofiler\_buffer\_category\_t::ROCProfiler\_BUFFER\_CATEGORY\_0 (C++ enumerator), 247  
rocprofiler\_buffer\_category\_t::ROCProfiler\_BUFFER\_CATEGORY\_1 (C++ enumerator), 247  
rocprofiler\_buffer\_category\_t::ROCProfiler\_BUFFER\_CATEGORY\_2 (C++ enumerator), 247  
rocprofiler\_buffer\_category\_t::ROCProfiler\_BUFFER\_CATEGORY\_3 (C++ enumerator), 247  
rocprofiler\_buffer\_category\_t::ROCProfiler\_BUFFER\_CATEGORY\_4 (C++ enumerator), 247  
rocprofiler\_buffer\_id\_t (C++ struct), 265  
rocprofiler\_buffer\_policy\_t (C++ enum), 259  
rocprofiler\_buffer\_policy\_t::ROCProfiler\_BUFFER\_POLICY\_DISABLE (C++ enumerator), 259  
rocprofiler\_buffer\_policy\_t::ROCProfiler\_BUFFER\_POLICY\_LAST (C++ enumerator), 259  
rocprofiler\_buffer\_policy\_t::ROCProfiler\_BUFFER\_POLICY\_LOSSY (C++ enumerator), 259  
rocprofiler\_buffer\_policy\_t::ROCProfiler\_BUFFER\_POLICY\_NONINTRUSIVE (C++ enumerator), 259  
rocprofiler\_buffer\_tracing\_cb\_t (C++ type), 199  
rocprofiler\_buffer\_tracing\_correlation\_id\_retirement\_record\_t (C++ struct), 204  
rocprofiler\_buffer\_tracing\_hip\_api\_ext\_record\_t (C++ struct), 203





rocprofiler\_callback\_tracing\_rccl\_api\_data\_t (C++ struct), 209

rocprofiler\_callback\_tracing\_record\_t (C++ struct), 265

rocprofiler\_callback\_tracing\_rocdecode\_api\_data\_t (C++ struct), 209

rocprofiler\_callback\_tracing\_rocjpeg\_api\_data\_t (C++ struct), 209

rocprofiler\_callback\_tracing\_runtime\_initialize\_t (C++ struct), 210

rocprofiler\_callback\_tracing\_scratch\_memory\_data\_t (C++ struct), 210

rocprofiler\_client\_finalize\_t (C++ type), 240

rocprofiler\_client\_id\_t (C++ struct), 243

ROC\_PROFILER\_CODE\_OBJECT\_ID\_NONE (C macro), 209

rocprofiler\_code\_object\_operation\_t (C++ enum), 256

rocprofiler\_code\_object\_operation\_t::ROC\_PROFILER\_CODE\_OBJECT\_OPERATION\_DEVICE\_KERNEL\_SYMBOL\_REGISTER (C++ enumerator), 256

rocprofiler\_code\_object\_operation\_t::ROC\_PROFILER\_CODE\_OBJECT\_OPERATION\_HOST\_KERNEL\_SYMBOL\_REGISTER (C++ enumerator), 256

rocprofiler\_code\_object\_operation\_t::ROC\_PROFILER\_CODE\_OBJECT\_OPERATION\_LOAD (C++ enumerator), 256

rocprofiler\_code\_object\_operation\_t::ROC\_PROFILER\_CODE\_OBJECT\_OPERATION\_STORE (C++ enumerator), 256

rocprofiler\_code\_object\_operation\_t::ROC\_PROFILER\_CODE\_OBJECT\_OPERATION\_NONE (C++ enumerator), 256

rocprofiler\_code\_object\_storage\_type\_t (C++ enum), 205

rocprofiler\_code\_object\_storage\_type\_t::ROC\_PROFILER\_CODE\_OBJECT\_STORAGE\_TYPE\_FILE (C++ enumerator), 205

rocprofiler\_code\_object\_storage\_type\_t::ROC\_PROFILER\_CODE\_OBJECT\_STORAGE\_TYPE\_FOR\_LAST (C++ enumerator), 205

rocprofiler\_code\_object\_storage\_type\_t::ROC\_PROFILER\_CODE\_OBJECT\_STORAGE\_TYPE\_MEMORY (C++ enumerator), 205

rocprofiler\_code\_object\_storage\_type\_t::ROC\_PROFILER\_CODE\_OBJECT\_STORAGE\_TYPE\_NONE (C++ enumerator), 205

rocprofiler\_configure (C++ function), 241

rocprofiler\_configure\_buffer\_dispatch\_counting\_service (C++ function), 217

rocprofiler\_configure\_buffer\_tracing\_service (C++ function), 201

rocprofiler\_configure\_callback\_dispatch\_counting\_service (C++ function), 218

rocprofiler\_configure\_callback\_tracing\_service (C++ function), 207

rocprofiler\_configure\_device\_counting\_service (C++ function), 215

rocprofiler\_configure\_device\_thread\_trace\_service (C++ function), 236

rocprofiler\_configure\_dispatch\_thread\_trace\_service (C++ function), 237

rocprofiler\_configure\_external\_correlation\_id\_request (C++ function), 220

rocprofiler\_configure\_func\_t (C++ type), 241

rocprofiler\_configure\_pc\_sampling\_service (C++ function), 228

rocprofiler\_context\_id\_t (C++ struct), 265

rocprofiler\_context\_is\_active (C++ function), 211

rocprofiler\_context\_is\_valid (C++ function), 211

ROC\_PROFILER\_CONTEXT\_NONE (C macro), 211

ROC\_PROFILER\_CORRELATION\_ID\_ANCESTOR\_NONE (C macro), 264

ROC\_PROFILER\_CORRELATION\_ID\_INTERNAL\_NONE (C macro), 264

rocprofiler\_correlation\_id\_t (C++ struct), 265

rocprofiler\_counter\_config\_id\_t (C++ struct), 265

rocprofiler\_counter\_dimension\_id\_t (C++ type), 263

rocprofiler\_counter\_dimension\_id\_t::ROC\_PROFILER\_COUNTER\_DIMENSION\_ID\_KERNEL\_SYMBOL\_REGISTER (C++ struct), 214

rocprofiler\_counter\_dimension\_id\_t::ROC\_PROFILER\_COUNTER\_DIMENSION\_ID\_HOST\_KERNEL\_SYMBOL\_REGISTER (C++ struct), 214

rocprofiler\_counter\_flag\_t::ROC\_PROFILER\_COUNTER\_FLAG\_APPEND (C++ enumerator), 262

rocprofiler\_counter\_flag\_t::ROC\_PROFILER\_COUNTER\_FLAG\_ASYNC (C++ enumerator), 262

rocprofiler\_counter\_flag\_t::ROC\_PROFILER\_COUNTER\_FLAG\_LAST (C++ enumerator), 262

rocprofiler\_counter\_flag\_t::ROC\_PROFILER\_COUNTER\_FLAG\_NONE (C++ enumerator), 262

rocprofiler\_counter\_id\_t (C++ struct), 265

rocprofiler\_counter\_info\_v1\_t (C++ struct), 214

rocprofiler\_counter\_info\_v1\_t::ROC\_PROFILER\_COUNTER\_INFO\_V1\_TYPE\_FILE (C++ struct), 214

rocprofiler\_counter\_info\_v1\_t::ROC\_PROFILER\_COUNTER\_INFO\_V1\_TYPE\_FOR\_LAST (C++ struct), 214

rocprofiler\_counter\_info\_v1\_t::ROC\_PROFILER\_COUNTER\_INFO\_V1\_TYPE\_MEMORY (C++ struct), 214

rocprofiler\_counter\_info\_v1\_t::ROC\_PROFILER\_COUNTER\_INFO\_V1\_TYPE\_NONE (C++ struct), 214

rocprofiler\_counter\_info\_version\_id\_t::ROC\_PROFILER\_COUNTER\_INFO\_VERSION\_ID\_FILE (C++ enumerator), 262

rocprofiler\_counter\_info\_version\_id\_t::ROC\_PROFILER\_COUNTER\_INFO\_VERSION\_ID\_MEMORY (C++ enumerator), 262

rocprofiler\_counter\_info\_version\_id\_t::ROC\_PROFILER\_COUNTER\_INFO\_VERSION\_ID\_NONE (C++ enumerator), 262

rocprofiler\_counter\_instance\_id\_t (C++ type), 266

rocprofiler\_counter\_record\_dimension\_info\_t (C++ struct), 266

rocprofiler\_counter\_record\_dimension\_instance\_info\_t (C++ struct), 214

rocprofiler\_counter\_record\_kind\_t (C++ enum), 262

rocprofiler\_counter\_record\_kind\_t::ROC\_PROFILER\_COUNTER\_RECORD\_KIND\_FILE (C++ enumerator), 262

rocprofiler\_counter\_record\_kind\_t::ROC\_PROFILER\_COUNTER\_RECORD\_KIND\_MEMORY (C++ enumerator), 262

rocprofiler\_counter\_record\_kind\_t::ROC\_PROFILER\_COUNTER\_RECORD\_KIND\_NONE (C++ enumerator), 262





(C++ enumerator), 226, 227

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_EXPORTER (C++ enumerator), 227

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_FLAT (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_COMP (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_LD58 (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_LDS\_DIRECT (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_MATRIX (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_MESSAGE (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_NO\_INST (C++ enumerator), 227

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_NONE (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_OTHER (C++ enumerator), 227

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTION\_TYPE\_SCALAR (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTIONS\_TYPE\_TEX (C++ enumerator), 226

rocprofiler\_pc\_sampling\_instruction\_type\_t::ROC\_PROFILER\_PC\_SAMPLING\_INSTRUCTIONS\_TYPE\_VALU (C++ enumerator), 226

rocprofiler\_pc\_sampling\_memory\_counters\_t (C++ struct), 232

rocprofiler\_pc\_sampling\_method\_t (C++ enum), 258

rocprofiler\_pc\_sampling\_method\_t::ROC\_PROFILER\_PC\_SAMPLING\_METHOD\_HOST\_TRAP (C++ enumerator), 258

rocprofiler\_pc\_sampling\_method\_t::ROC\_PROFILER\_PC\_SAMPLING\_METHOD\_KERNEL\_STACK (C++ enumerator), 258

rocprofiler\_pc\_sampling\_method\_t::ROC\_PROFILER\_PC\_SAMPLING\_METHOD\_NONE (C++ enumerator), 258

rocprofiler\_pc\_sampling\_method\_t::ROC\_PROFILER\_PC\_SAMPLING\_METHOD\_STOCHASTIC (C++ enumerator), 258

rocprofiler\_pc\_sampling\_record\_host\_trap\_v0\_t (C++ struct), 231

rocprofiler\_pc\_sampling\_record\_invalid\_t (C++ struct), 232

rocprofiler\_pc\_sampling\_record\_kind\_t (C++ enum), 263

rocprofiler\_pc\_sampling\_record\_kind\_t::ROC\_PROFILER\_PC\_SAMPLING\_RECORD\_HOST\_TRAP\_V0\_SAMPLE (C++ enumerator), 263

rocprofiler\_pc\_sampling\_record\_kind\_t::ROC\_PROFILER\_PC\_SAMPLING\_RECORD\_INVALID\_SAMPLE (C++ enumerator), 263

rocprofiler\_pc\_sampling\_record\_kind\_t::ROC\_PROFILER\_PC\_SAMPLING\_RECORD\_INVALID\_SAMPLE (C++ enumerator), 263

rocprofiler\_pc\_sampling\_record\_kind\_t::ROC\_PROFILER\_PC\_SAMPLING\_RECORD\_INVALID\_SAMPLE (C++ enumerator), 263

rocprofiler\_pc\_sampling\_record\_kind\_t::ROC\_PROFILER\_PC\_SAMPLING\_RECORD\_NONE (C++ enumerator), 263

(C++ enumerator), 263

(C++ enumerator), 263

(C++ struct), 231

(C++ struct), 232

(C++ struct), 232

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_LDS8

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_LDS\_DIRECT

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_MATRIX

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_MESSAGE

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_NO\_INST

rocprofiler\_pc\_sampling\_unit\_t::ROC\_PROFILER\_PC\_SAMPLING\_UNIT\_NONE

rocprofiler\_pc\_t (C++ struct), 231

(C++ function), 222

(C++ function), 221

(C++ function), 198

(C++ type), 198

rocprofiler\_query\_buffer\_tracing\_kind\_name (C++ function), 201

rocprofiler\_query\_buffer\_tracing\_kind\_operation\_name (C++ function), 202

rocprofiler\_query\_buffer\_tracing\_kind\_name (C++ function), 207

rocprofiler\_query\_buffer\_tracing\_kind\_operation\_name (C++ function), 208

rocprofiler\_query\_info (C++ function), 213

rocprofiler\_query\_info (C++ function), 221

rocprofiler\_query\_intercept\_table\_name (C++ function), 223

rocprofiler\_query\_pc\_sampling\_agent\_configurations (C++ function), 230

rocprofiler\_query\_record\_counter\_id (C++ function), 212

rocprofiler\_query\_record\_host\_trap\_v0\_sample (C++ function), 212

rocprofiler\_record\_counter\_t (C++ type), 264

rocprofiler\_record\_counter\_t (C++ type), 264

rocprofiler\_record\_counter\_t (C++ type), 264

compute\_hash (C++

*function*), 264  
 rocprofiler\_record\_header\_t (C++ *struct*), 265  
 rocprofiler\_runtime\_initialization\_operation\_t (C++ *enum*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_HIP (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_HSA (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_LAST (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_MARKER (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_NONE (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_RCCL (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_ROCDECODE (C++ *enumerator*), 261  
 rocprofiler\_runtime\_initialization\_operation\_t::ROCPROFILER\_RUNTIME\_INITIALIZATION\_ROCJPEG (C++ *enumerator*), 261  
 rocprofiler\_runtime\_library\_t (C++ *enum*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_HIP\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_HSA\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_LIBRARY\_ASIO (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_MARKER\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_RCCL\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_ROCDECODE\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_runtime\_library\_t::ROCPROFILER\_ROCJPEG\_RUNTIME\_LIBRARY (C++ *enumerator*), 260  
 rocprofiler\_sample\_device\_counting\_service (C++ *function*), 216  
 rocprofiler\_scratch\_memory\_operation\_t (C++ *enum*), 259  
 rocprofiler\_scratch\_memory\_operation\_t::ROCPROFILER\_SCRATCH\_MEMORY\_ASYNC (C++ *enumerator*), 259  
 rocprofiler\_scratch\_memory\_operation\_t::ROCPROFILER\_SCRATCH\_MEMORY\_ASYNC\_BLOCKING (C++ *enumerator*), 259  
 rocprofiler\_scratch\_memory\_operation\_t::ROCPROFILER\_SCRATCH\_MEMORY\_FREE (C++ *enumerator*), 259  
 rocprofiler\_scratch\_memory\_operation\_t::ROCPROFILER\_SCRATCH\_MEMORY\_INVALID (C++ *enumerator*), 259  
 rocprofiler\_scratch\_memory\_operation\_t::ROCPROFILER\_SCRATCH\_MEMORY\_NONE (C++ *enumerator*), 259  
 rocprofiler\_start\_context (C++ *function*), 211  
 rocprofiler\_status\_t (C++ *enum*), 244  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AGENT\_ARCH (C++ *enumerator*), 244  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AGENT\_DISPATCH (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AGENT\_MISMATCH (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AGENT\_NOT\_FOUND (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AQL\_NO\_EVENTS (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AST\_GENERATION (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_AST\_NOT\_FOUND (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_BUFFER\_BUSY (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_BUFFER\_NOT\_AVAILABLE (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONFIGURATION (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_COMPLETION (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_ERROR (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_ID (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_INVALID (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_NOT\_AVAILABLE (C++ *enumerator*), 244  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_CONTEXT\_NOT\_READY (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_COUNTER\_NOT\_AVAILABLE (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_DIM\_NOT\_FOUND (C++ *enumerator*), 246  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_EXCEEDS\_HW\_LIMITS (C++ *enumerator*), 247  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_FINALIZED (C++ *enumerator*), 246  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_HSA\_NOT\_LOADED (C++ *enumerator*), 246  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_INCOMPATIBLE\_ATTRIBUTES (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_INCOMPATIBLE\_OPERATION (C++ *enumerator*), 246  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_INVALID\_ARCH (C++ *enumerator*), 245  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_KIND\_NOT\_FOUND (C++ *enumerator*), 244  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_METRIC\_NOT\_AVAILABLE (C++ *enumerator*), 246  
 rocprofiler\_status\_t::ROCPROFILER\_STATUS\_ERROR\_NO\_HARDWARE (C++ *enumerator*), 246

(C++ enumerator), 247  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_NO\_PROFILE\_QUEUE (C++ enumerator), 233  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_NOT\_AVAILABLE (C++ enumerator), 233  
 (C++ enumerator), 247  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_NOT\_IMPLEMENTED (C++ enumerator), 233  
 (C++ enumerator), 245  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_OPERATION\_NOT\_FOUND (C++ enumerator), 233  
 (C++ enumerator), 245  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_OUT\_OF\_RESOURCES (C++ enumerator), 233  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_PERMISSION\_DENIED (C++ enumerator), 233  
 (C++ enumerator), 247  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_PROFILE\_COUNTER\_NOT\_FOUND (C++ enumerator), 246  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_PROFILE\_NOT\_FOUND (C++ enumerator), 234  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_SAMPLE\_RATE\_EXCEEDED (C++ enumerator), 234  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_SERVICE\_ALREADY\_CONFIGURED (C++ enumerator), 245  
 (C++ enumerator), 245  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_ERROR\_THREAD\_NOT\_FOUND (C++ enumerator), 234  
 (C++ enumerator), 245  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_INTERNAL\_NO\_AGENT\_CONTEXT (C++ enumerator), 234  
 (C++ enumerator), 246  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_LAST (C++ enumerator), 234  
 (C++ enumerator), 247  
 rocprofiler\_status\_t::ROCProfiler\_STATUS\_SUCCESS (C++ enumerator), 234  
 (C++ enumerator), 244  
 rocprofiler\_stop\_context (C++ function), 211  
 rocprofiler\_stream\_id\_t (C++ struct), 265  
 rocprofiler\_thread\_id\_t (C++ type), 263  
 rocprofiler\_thread\_trace\_control\_flags\_t (C++ enum), 233  
 rocprofiler\_thread\_trace\_control\_flags\_t::ROCProfiler\_THREAD\_TRACE\_CONTROL\_DONE (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_control\_flags\_t::ROCProfiler\_THREAD\_TRACE\_CONTROL\_START\_AND\_STOP (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_callback\_t (C++ type), 236  
 rocprofiler\_thread\_trace\_decoder\_codeobj\_load (C++ function), 238  
 rocprofiler\_thread\_trace\_decoder\_codeobj\_unload (C++ function), 238  
 rocprofiler\_thread\_trace\_decoder\_create (C++ function), 237  
 rocprofiler\_thread\_trace\_decoder\_destroy (C++ function), 238  
 rocprofiler\_thread\_trace\_decoder\_id\_t (C++ struct), 239  
 rocprofiler\_thread\_trace\_decoder\_info\_string (C++ function), 239  
 rocprofiler\_thread\_trace\_decoder\_info\_t (C++ enum), 233  
 rocprofiler\_thread\_trace\_decoder\_info\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INFO\_THREAD\_ID (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_info\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INFO\_THREAD\_NAME (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_info\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INFO\_THREAD\_PRIORITY (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_info\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INFO\_THREAD\_TYPE (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_category\_t (C++ enum), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_category\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INST\_CATEGORY\_THREAD\_ID (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_category\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INST\_CATEGORY\_THREAD\_NAME (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_category\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INST\_CATEGORY\_THREAD\_PRIORITY (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_category\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_INST\_CATEGORY\_THREAD\_TYPE (C++ enumerator), 233  
 rocprofiler\_thread\_trace\_decoder\_inst\_t (C++ struct), 240  
 rocprofiler\_thread\_trace\_decoder\_occupancy\_t (C++ struct), 239  
 rocprofiler\_thread\_trace\_decoder\_pc\_t (C++ struct), 239  
 rocprofiler\_thread\_trace\_decoder\_perfevent\_t (C++ struct), 239  
 rocprofiler\_thread\_trace\_decoder\_realtime\_t (C++ struct), 240  
 rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enum), 235  
 rocprofiler\_thread\_trace\_decoder\_record\_type\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_TYPE\_THREAD\_ID (C++ enumerator), 235  
 rocprofiler\_thread\_trace\_decoder\_record\_type\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_TYPE\_THREAD\_NAME (C++ enumerator), 235  
 rocprofiler\_thread\_trace\_decoder\_record\_type\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_TYPE\_THREAD\_PRIORITY (C++ enumerator), 235  
 rocprofiler\_thread\_trace\_decoder\_record\_type\_t::ROCProfiler\_THREAD\_TRACE\_DECODER\_RECORD\_TYPE\_THREAD\_TYPE (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 232

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 232

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 232

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 232

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 232

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ type), 235

rocprofiler\_thread\_trace\_decoder\_record\_type\_t (C++ enumerator), 235

rocprofiler\_thread\_trace\_decoder\_shaderdata\_flags\_t (C++ enum), 234

rocprofiler\_thread\_trace\_decoder\_shaderdata\_flags\_t (C++ enumerator), 234

rocprofiler\_thread\_trace\_decoder\_shaderdata\_t (C++ struct), 240

rocprofiler\_thread\_trace\_decoder\_wave\_state\_t (C++ struct), 239

rocprofiler\_thread\_trace\_decoder\_wave\_t (C++ struct), 240

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enum), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_decoder\_wstate\_type\_t (C++ enumerator), 233

rocprofiler\_thread\_trace\_dispatch\_callback\_t (C++ type), 236

rocprofiler\_thread\_trace\_parameter\_t (C++ struct), 239

rocprofiler\_thread\_trace\_parameter\_type\_t (C++ enum), 232

rocprofiler\_thread\_trace\_parameter\_type\_t::ROCProfiler\_THREAD\_TRACE\_PARAMETER\_BUFFER\_SIZE (C++ enumerator), 232

rocprofiler\_thread\_trace\_parameter\_type\_t::ROCProfiler\_THREAD\_TRACE\_PARAMETER\_LAST (C++ enumerator), 232

rocprofiler\_thread\_trace\_parameter\_type\_t::ROCProfiler\_THREAD\_TRACE\_PARAMETER\_NO\_DETAIL (C++ enumerator), 232

rocprofiler\_thread\_trace\_parameter\_type\_t::ROCProfiler\_THREAD\_TRACE\_PARAMETER\_PERFCOUNTER (C++ enumerator), 232