Rocprofiler SDK

Release 0.5.0

Advanced Micro Devices, Inc.

Dec 05, 2024

INSTALL

ROCprofiler-SDK is a tooling infrastructure for profiling general-purpose GPU compute applications running on the ROCm software. It supports application tracing to provide a big picture of the GPU application execution and kernel profiling 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 and hosted at https://github.com/ROCm/rocprofiler-sdk.

Note

ROCprofiler-SDK is in beta and subject to change in future releases.

The documentation is structured as follows:

Install

- *Installation*

How to

- *Using rocprofv3*
- *Samples*

API reference

- *Buffered services*
- *Callback services*
- *Counter collection services*
- *Intercept table*
- *PC sampling*
- *Tool library*
- *API library*

Conceptual

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

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

You can find licensing information on the Licensing page.

CHAPTER

ONE

ROCPROFILER-SDK INSTALLATION

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

1.1 Supported systems

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

- Ubuntu 20.04
- Ubuntu 22.04
- OpenSUSE 15.4
- RedHat 8.8

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

1.1.1 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`.

1.2 Build requirements

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).

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

1.3 Building ROCprofiler-SDK

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

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

1.4 Installing ROCprofiler-SDK

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

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

1.5 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
```

CHAPTER

TWO

USING ROCPROFV3

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

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

`rocprofv3` 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
```

Before you start tracing or profiling your HIP application using `rocprofv3`, build the application using:

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

2.1 Options

Here is the sample of commonly used `rocprofv3` command-line options. Some options are used for application tracing and some for kernel profiling while the output control options control the presentation and redirection of the generated output.

Table 2.1: rocprofv3 options

Option	Description	Use
-i | --input	Specifies the input file. JSON and YAML formats support configuration of all command-line options whereas the text format only supports specifying HW counters.	Run Configuration
-d | --output-directory	Specifies the path for the output files. Supports special keys: %hostname%, %pid%, %rank%, etc.	Output control
-o | --output-file	Specifies the name of the output file. Note that this name is appended to the default names (_api_trace or counter_collection.csv) of the generated files'. Supports special keys: %hostname%, %pid%, %rank%, etc.	Output control
--output-format	For adding output format (supported formats: csv, json, pftrace)	Output control
-r | --runtime-trace	Collects HIP (runtime), memory copy, marker, scratch memory, and kernel dispatch traces.	Application Tracing
-s | --sys-trace	Collects HIP, HSA, memory copy, marker, scratch memory, and kernel dispatch traces.	Application Tracing
--hip-trace	Collects HIP runtime and compiler traces.	Application tracing
--kernel-trace	Collects kernel dispatch traces.	Application tracing
--marker-trace	Collects marker (ROC-TX) traces.	Application tracing
--memory-copy-trace	Collects memory copy traces.	Application tracing
--scratch-memory-tr	Collects scratch memory operations traces.	Application tracing
--hsa-trace	Collects HSA API traces.	Application tracing
--hip-runtime-trace	Collects HIP runtime API traces.	Application tracing
--hsa-core-trace	Collects HSA API traces (core API).	Application tracing
--hsa-amd-trace	Collects HSA API traces (AMD-extension API).	Application tracing
--stats	For Collecting statistics of enabled tracing types	Application tracing
-p | --summary	Display summary of collected data	Application tracing
--kernel-include-re	Include the kernels matching this filter.	Kernel Dispatch Counter Collection
--kernel-exclude-re	Exclude the kernels matching this filter.	Kernel Dispatch Counter Collection
--kernel-iteration-	Iteration range for each kernel that match the filter [start-stop].	Kernel Dispatch Counter Collection
-L | --list-metrics	List metrics for counter collection.	Kernel Dispatch Counter Collection
-M | --mangled-kernels	Overrides the default demangling of kernel names.	Output control
-T | --truncate-kernels	Truncates the demangled kernel names for improved readability.	Output control
--output-format	For adding output format (supported formats: csv, json, pftrace, otf2)	Output con-

To see exhaustive list of `rocprofv3` options, run:

```
rocprofv3 --help
```

2.2 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>
```

2.2.1 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.

To trace HIP runtime APIs, use:

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

The above 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 2.2: HIP runtime api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_COMI	__hipRegisterFatBinary	208	208	1	1508780270085955	1508780270096795
HIP_COMI	__hipRegisterFunction	208	208	2	1508780270104242	1508780270115355
HIP_COMI	__hipPushCallConfiguration	208	208	3	1508780613897816	1508780613898701
HIP_COMI	__hipPopCallConfiguration	208	208	4	1508780613901714	1508780613902200

To trace HIP compile time APIs, use:

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

The above 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 2.3: HIP compile time api trace

Domain	Function	Process_Id	Thread_Id	Correlation_Id	Start_Timestamp	End_Timestamp
HIP_COMI	__hipRegisterFatBinary	208	208	1	1508780270085955	1508780270096795
HIP_COMI	__hipRegisterFunction	208	208	2	1508780270104242	1508780270115355
HIP_COMI	__hipPushCallConfiguration	208	208	3	1508780613897816	1508780613898701
HIP_COMI	__hipPopCallConfiguration	208	208	4	1508780613901714	1508780613902200

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

2.2.2 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 -- <application_path>
```

The above 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 2.4: HSA api trace

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

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

2.2.3 Marker trace

In certain situations, such as debugging performance issues in large-scale GPU programs, API-level tracing might be too fine-grained to provide a big picture 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. As the scope of code for instrumentation is defined using the enclosing API calls, it is called a range. A range is a programmer-defined task that has a well-defined start and end code scope. You can also refine the scope specified within a range using further nested ranges. `rocprofv3` also reports the timelines for these nested ranges.

Here is a 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`: Request any currently running profiling tool that it should stop collecting data.
- `roctxProfilerResume`: Request any currently running profiling tool that it should resume collecting data.
- `roctxGetThreadId`: 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*)*.
- `roctxNameOsThread`: Current CPU OS thread to be labeled by the provided name in the output of the profiling tool.
- `roctxNameHsaAgent`: Given HSA agent to be labeled by the provided name in the output of the profiling tool.
- `roctxNameHipDevice`: Given HIP device id to be labeled by the provided name in the output of the profiling tool.
- `roctxNameHipStream`: Given HIP stream to be labeled by the provided name in the output of the profiling tool.

Note

To use `rocprofv3` for marker tracing, including and linking to old ROCTx works but it is recommended to switch to new ROCTx because it has been extended with new APIs. To use new ROCTx, please include header "`rocprofiler-sdk-roctx/roctx.h`" and link your application with `librocprofiler-sdk-roctx.so`. Above list of APIs is not exhaustive. See public header file "`rocprofiler-sdk-roctx/roctx.h`" for full list.

See how to use `ROCTx` APIs in the MatrixTranspose application below:

```
#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 -- <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 2.5: Marker api trace

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

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

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

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

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

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

// Lauching 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));
```

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

The above command generates a ``hip_api_trace.csv`` file prefixed with the process ID,␣
↪which has only 2  `hipMemcpy` calls and the in between ``hipMemcpyDeviceToHost`` is␣
↪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
```

2.2.4 Kernel Rename

To rename kernels with their enclosing roctxRangePush/roctxRangePop message. Known as –roctx-rename in earlier rocprof versions.

See how to use `--kernel-rename` option with help of below code snippet:

```
#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 -- <application_path>
```

The above 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","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
```

2.2.5 Kokkos Trace

rocprofv3 has a built-in Kokkos Tools library support to trace Kokkos API calls. Kokkos is a C++ library for writing performance portable applications. It is used in many scientific applications to write performance portable code that can run on CPUs, GPUs, and other accelerators. rocprofv3 loads a built-in Kokkos tools library which emits roctx ranges with the labels passed through the API, e.g. Kokkos::parallel_for("MyParallelForLabel", ...); will internally calls for roctxRangePush and enables the kernel renaming option so that the highly templated kernel names are replaced by the Kokkos labels. To enable built-in marker support, use the `kokkos-trace` option. Internally this option enables `marker-trace` and `kernel-rename`.:

```
rocprofv3 --kokkos-trace -- <application_path>
```

The above 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
```

2.2.6 Kernel trace

To trace kernel dispatch traces, use:

```
rocprofv3 --kernel-trace -- <application_path>
```

The above 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 2.6: Kernel trace

Kinc	Age	Que	Thre	Dis-patc	Ker-nel_	Ker-nel_	Cor-re-la-tion_	Start_Tin	End_Tim	Pri-vate	Gro	Wor grou	Wor grou	Wor grou	Gric	Gric	Grid_Size_Z
KEF NEL	1	1	69	1	16	void ad-di-tion_ float cons float cons int, int)	1451	88193302	88193302	0	0	64	1	1	1024	1024	1
KEF NEL	1	2	69	5	16	void ad-di-tion_ float cons float cons int, int)	1484	88193302	88193302	0	0	64	1	1	1024	1024	1
KEF NEL	1	1	69	2	19	sub-tract float cons float cons int, int)	1459	88193302	88193302	0	0	64	1	1	1024	1024	1
KEF NEL	1	3	69	9	16	void ad-di-tion_ float cons float cons int, int)	1517	88193302	88193302	0	0	64	1	1	1024	1024	1
KEF NEL	1	4	69	13	16	void ad-di-tion_ float cons float cons int, int)	1550	88193302	88193302	0	0	64	1	1	1024	1024	1
KEF NEL	1	2	69	6	19	sub-tract float cons float cons	1492	88193302	88193302	0	0	64	1	1	1024	1024	1

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

2.2.7 Memory copy trace

To trace memory moves across the application, use:

```
rocprofv3 --memory-copy-trace -- <application_path>
```

The above 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 2.7: Memory copy trace

Kind	Direction	Source_Ag	Destina-tion_Agent	Correla-tion_Id	Start_Timestamp	End_Timestamp
MEM-ORY_COPY	MEM-ORY_COPY	0	1	0	14955949675563	14955950239443
MEM-ORY_COPY	MEM-ORY_COPY	1	0	0	14955952733485	14955953315285

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

2.2.8 Runtime trace

This is a short-hand option which attempts to target the most relevant tracing options for a standard user by excluding tracing the 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 not relevant to most users. The HIP compiler API is excluded because these are functions which are automatically inserted during HIP compilation and thus, also tend to be implementation details which are not relevant to most users.

At present, *–runtime-trace* enables tracing the HIP runtime API, the marker API, kernel dispatches, and memory operations (copies and scratch).

```
rocprofv3 --runtime-trace -- <application_path>
```

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

2.2.9 System trace

This is an all-inclusive option to collect all the above-mentioned traces.

```
rocprofv3 --sys-trace -- <application_path>
```

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

2.2.10 Scratch memory trace

This option collects scratch memory operation's traces. Scratch is an address space on AMD GPUs, which is 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. With this option, you can trace when the `rocr` runtime allocates, frees, and tries to reclaim scratch memory.

```
rocprofv3 --scratch-memory-trace -- <application_path>
```

2.2.11 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 -- <application_path>
```

The above 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 2.8: RCCL trace

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

2.2.12 Post-processing tracing options

2.2.13 1. Stats

This option collects statistics for the enabled tracing types. For example, to collect statistics of HIP APIs, when HIP trace is enabled. A higher percentage in statistics can help user focus on the API/function that has taken the most time:

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

The above 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 2.9: HIP stats

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

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

Table 2.10: Domain stats

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

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

2.2.14 2. Summary

Output single summary of tracing data at the 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   |      0.066695 |            83953 |           132851 |        3.458e+04 |
| hipMalloc                          | HIP_API      |               2 |           51981 |    2.599e+04   |      0.015991 |            10502 |            41479 |        2.190e+04 |
```

2.2.15 2.1 Summary per domain

Outputs the summary of each tracing domain at the end of profiling session.

```
rocprofv3 -D --hsa-trace --hip-trace  -- <application_path>
```

The above command generates a `hip_trace.csv`, `hsa_trace.csv` file prefixed with the process ID along with the summary of each domain at the terminal.

2.2.16 2.2 Summary groups

Users can create a summary of multiple domains by specifying the domain names in the command line. The summary groups are separated by a pipe (|) symbol. To create 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 create a summary for `MEMORY_COPY` and `HIP_API` domains, use:

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

```
ROCPROFV3 HIP_API + MEMORY_COPY SUMMARY:
|                NAME                |    DOMAIN    |      CALLS      | DURATION (nsec) | AVERAGE (nsec) | PERCENT (INC) |    MIN (nsec)    |    MAX (nsec)    |      STDDEV      |
|------------------------------------|--------------|-----------------|-----------------|----------------|---------------|------------------|------------------|------------------|
| hipMemcpy                          | HIP_API      |               2 |       228708882 |    1.144e+08   |     70.476616 |           201413 |        228507469 |        1.614e+08 |
| hipGetDevicePropertiesR0600        | HIP_API      |               1 |        94988182 |    9.499e+07   |     29.270597 |         94988182 |         94988182 |        0.000e+00 |
| hipLaunchKernel                    | HIP_API      |               1 |          338689 |    3.387e+05   |      0.104367 |           338689 |           338689 |        0.000e+00 |
| hipFree                            | HIP_API      |               2 |          197402 |    9.870e+04   |      0.060829 |            82090 |           115312 |        2.349e+04 |
| MEMORY_COPY_DEVICE_TO_HOST         | MEMORY_COPY  |               1 |          109034 |    1.090e+05   |      0.033599 |           109034 |           109034 |        0.000e+00 |
| MEMORY_COPY_HOST_TO_DEVICE         | MEMORY_COPY  |               1 |          102059 |    1.021e+05   |      0.031449 |           102059 |           102059 |        0.000e+00 |
| hipMalloc                          | HIP_API      |               2 |           64612 |    3.231e+04   |      0.019910 |            20713 |            43899 |        1.639e+04 |
| __hipRegisterFatBinary             | HIP_API      |               1 |            4124 |    4.124e+03   |      0.001271 |             4124 |             4124 |        0.000e+00 |
| __hipRegisterFunction              | HIP_API      |               1 |            2877 |    2.877e+03   |      0.000887 |             2877 |             2877 |        0.000e+00 |
| __hipPushCallConfiguration         | HIP_API      |               1 |             788 |    7.880e+02   |      0.000243 |              788 |              788 |        0.000e+00 |
| __hipPopCallConfiguration          | HIP_API      |               1 |             753 |    7.530e+02   |      0.000232 |              753 |              753 |        0.000e+00 |
```

2.3 Kernel profiling

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 profiling 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.

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

2.3.1 Input file

To collect the desired basic counters or derived metrics or tracing, mention them in an input file. The input file could be in text (.txt), yaml (.yaml/.yml), or JSON (.json) format.

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

The JSON and YAML files supports all the command line options and it can be used to configure both tracing and profiling. The input file has an array of profiling/tracing configurations called jobs. Each job is used to configure profiling/tracing for an application execution. The input schema of these files is given below.

2.3.2 Properties

- ``**jobs**`` *(array)*: rocprofv3 input data per application run.
  - **Items** *(object)*: data for rocprofv3.
    - ``**pmc**`` *(array)*: list of counters to collect.
    - ``**kernel_include_regex**`` *(string)*: Include the kernels matching this filter.
    - ``**kernel_exclude_regex**`` *(string)*: Exclude the kernels matching this filter.
    - ``**kernel_iteration_range**`` *(string)*: Iteration range for each kernel that match the filter [start-stop].
    - ``**hip_trace**`` *(boolean)*: For Collecting HIP Traces (runtime + compiler).
    - ``**hip_runtime_trace**`` *(boolean)*: For Collecting HIP Runtime API Traces.
    - ``**hip_compiler_trace**`` *(boolean)*: For Collecting HIP Compiler generated code Traces.
    - ``**marker_trace**`` *(boolean)*: For Collecting Marker (ROCTx) Traces.
    - ``**kernel_trace**`` *(boolean)*: For Collecting Kernel Dispatch Traces.
    - ``**memory_copy_trace**`` *(boolean)*: For Collecting Memory Copy Traces.
    - ``**scratch_memory_trace**`` *(boolean)*: For Collecting Scratch Memory operations Traces.
    - ``**stats**`` *(boolean)*: For Collecting statistics of enabled tracing types.
    - ``**hsa_trace**`` *(boolean)*: For Collecting HSA Traces (core + amd + image + finalizer).
    - ``**hsa_core_trace**`` *(boolean)*: For Collecting HSA API Traces (core API).
    - ``**hsa_amd_trace**`` *(boolean)*: For Collecting HSA API Traces (AMD-extension API).
    - ``**hsa_finalize_trace**`` *(boolean)*: For Collecting HSA API Traces (Finalizer-extension API).
    - ``**hsa_image_trace**`` *(boolean)*: For Collecting HSA API Traces (Image-extension API).

- ``**sys_trace**`` *(boolean)*: For Collecting HIP, HSA, Marker (ROCTx), Memory copy, Scratch memory, and Kernel dispatch traces.
- ``**mangled_kernels**`` *(boolean)*: Do not demangle the kernel names.
- ``**truncate_kernels**`` *(boolean)*: Truncate the demangled kernel names.
- ``**output_file**`` *(string)*: For the output file name.
- ``**output_directory**`` *(string)*: For adding output path where the output files will be saved.
- ``**output_format**`` *(array)*: For adding output format (supported formats: csv, json, pftrace, otf2).
- ``**list_metrics**`` *(boolean)*: List the metrics.
- ``**log_level**`` *(string)*: fatal, error, warning, info, trace.
- ``**preload**`` *(array)*: Libraries to prepend to LD_PRELOAD (usually for sanitizers).

```
$ cat input.txt

pmc: GPUBusy SQ_WAVES
pmc: GRBM_GUI_ACTIVE
```

```
$ 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
    ]
}
```

```
  $ cat input.yaml

jobs:
  - pmc:
      - SQ_WAVES
      - GRBM_COUNT
      - GRBM_GUI_ACTIVE
      - 'TCC_HIT[1]'
      - 'TCC_HIT[2]'
  - pmc:
      - FETCH_SIZE
      - WRITE_SIZE
```

2.3.3 Command-line

Desired counters can now be collected as `command-line` option as well.

To supply the counters via `command-line` options, use:

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

Note

1. Please note that more than 1 counters should be separated by a space or a comma.
2. Job will fail if entire set of counters cannot be collected in single pass

2.3.4 Kernel profiling output

To supply the input file for kernel profiling, use:

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

Running the above command 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.

In case of JSON or YAML input file, for each job, a directory `pass_n` containing a `counter_collection.csv` file is generated where n = 1…N jobs.

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 2.11: Counter collection

Cor-relation_	Dis-patch	Agen	Queu	Pro-cess	Thre	Grid_	Ker-nel_	Ker-nel_	Work-grou	LDS_	Scra	VGP	SGP	Coun	Coun	Start	End_	Time-stamp
1	1	1	1	1939	1939	1048	16	void ad-di-tion_ float const float const int, int)	64	0	0	8	16	SQ_W	1638	2228	2228	19754
2	2	1	1	1939	1939	1048	19	sub-tract_ float const float const int, int)	64	0	0	8	16	SQ_W	1638	2228	2228	19754
5	5	1	2	1939	1939	1048	16	void ad-di-tion_ float const float const int, int)	64	0	0	8	16	SQ_W	1638	2228	2228	19754
9	9	1	3	1939	1939	1048	16	void ad-di-tion_ float const float const int, int)	64	0	0	8	16	SQ_W	1638	2228	2228	19754
13	13	1	4	1939	1939	1048	16	void ad-di-tion_ float const float const int, int)	64	0	0	8	16	SQ_W	1638	2228	2228	19754
3	3	1	1	1939	1939	1048	17	mul-ti-ply_k float const float	64	0	0	8	16	SQ_W	1638	2228	2228	19754

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

2.3.5 Kernel filtering

rocprofv3 supports kernel filtering in case of profiling. A kernel filter is a set of a regex string (to include the kernels matching this filter), a regex string (to exclude the kernels matching this filter), and an iteration range (set of iterations of the included kernels). If the iteration range is not provided then all iterations of the included kernels are profiled.

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

2.3.6 Agent info

Note

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

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

```
$ cat 238_agent_info.csv

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

2.3.7 Kernel filtering

Kernel filtering allows you to filter the kernel profiling output based on the kernel name by specifying regex strings in the input file. To include kernel names matching the regex string in the kernel profiling output, use `kernel_include_regex`. To exclude the kernel names matching the regex string from the kernel profiling output, use `kernel_exclude_regex`. You can also specify an iteration range for set of iterations of the included kernels. If the iteration range is not specified, then all iterations of the included kernels are profiled.

Here is an input file with kernel filters:

```
$ cat input.yml
jobs:
    - pmc: [SQ_WAVES]
    kernel_include_regex: "divide"
    kernel_exclude_regex: ""
```

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

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

$ cat pass_1/312_counter_collection.csv
"Correlation_Id","Dispatch_Id","Agent_Id","Queue_Id","Process_Id","Thread_Id","Grid_Size
↪","Kernel_Name","Workgroup_Size","LDS_Block_Size","Scratch_Size","VGPR_Count","SGPR_
↪Count","Counter_Name","Counter_Value","Start_Timestamp","End_Timestamp"
4,4,1,1,36499,36499,1048576,"divide_kernel(float*, float const*, float const*, int, int)
↪",64,0,0,12,16,"SQ_WAVES",16384,2228955885095594,2228955885119754
8,8,1,2,36499,36499,1048576,"divide_kernel(float*, float const*, float const*, int, int)
↪",64,0,0,12,16,"SQ_WAVES",16384,2228955885095594,2228955885119754
12,12,1,3,36499,36499,1048576,"divide_kernel(float*, float const*, float const*, int,␣
↪int)",64,0,0,12,16,"SQ_WAVES",16384,2228955892986914,2228955893006114
16,16,1,4,36499,36499,1048576,"divide_kernel(float*, float const*, float const*, int,␣
↪int)",64,0,0,12,16,"SQ_WAVES",16384,2228955892986914,2228955893006114
```

2.4 Output file fields

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

Table 2.12: output file fields

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

2.5 Output formats

`rocprofv3` supports the following output formats:

- CSV (Default)
- 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)

You can specify the output format using the `--output-format` command-line option. Format selection is case-insensitive and multiple output formats are supported. For example: `--output-format json` enables JSON output exclusively whereas `--output-format csv json pftrace otf2` enables all four output formats for the run.

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

For .otf2 trace visualization, open the trace in vampir.eu or any supported visualizer.

> **Note**
>
> For large trace files(> 10GB), its recommended to use otf2 format.

2.5.1 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. The Perfetto UI does not accept this 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. You can generate the JSON output using `--output-format json` command-line option.

2.5.2 Properties

- **`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.
* **`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 VGPRs.

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

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

* **`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.

CHAPTER

THREE

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
```

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
```

CHAPTER

FOUR

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.

4.1 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.

4.1.1 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.

4.1.2 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);
}
```

4.1.3 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,
                                             size_t                           ␣
```

(continues on next page)

(continued from previous page)

```
↪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. ...
}
```

4.2 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"};
        }
    }
}
```

4.3 Buffer tracing record

Unlike callback tracing records, there is no common set of data for each buffer tracing record. However, many buffer tracing records contain a `kind` and an `operation` field. You can obtain the value for the `kind` of tracing using `rocprofiler_query_buffer_tracing_kind_name` function and the value for the `operation` specific to a tracing kind using the `rocprofiler_query_buffer_tracing_kind_operation_name` function. You can also iterate over all the buffer tracing `kinds` and `operations` for each tracing kind using the `rocprofiler_iterate_buffer_tracing_kinds` and `rocprofiler_iterate_buffer_tracing_kind_operations` functions.

The buffer tracing record data types are available in the rocprofiler-sdk/buffer_tracing.h header.

CHAPTER

FIVE

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.

5.1 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. ...
}
```

5.2 Callback tracing callback function

Here is the callback tracing callback function:

```
typedef void (*rocprofiler_callback_tracing_cb_t)(rocprofiler_callback_tracing_record_t␣
↪record,
```

(continues on next page)

(continued from previous page)

```
                                                rocprofiler_user_data_t*                ␣
↪user_data,
                                                void* callback_data)
```

The parameters `record` and `user_data` are discussed here:

- `record`: Contains the information to uniquely identify a tracing record type. Here is the definition:

  ```
  typedef struct rocprofiler_callback_tracing_record_t
  {
    rocprofiler_context_id_t            context_id;
    rocprofiler_thread_id_t             thread_id;
    rocprofiler_correlation_id_t        correlation_id;
    rocprofiler_callback_tracing_kind_t kind;
    uint32_t                            operation;
    rocprofiler_callback_phase_t        phase;
    void*                               payload;
  } rocprofiler_callback_tracing_record_t;
  ```

  The underlying type of `payload` field is typically unique to a domain and, less frequently, an operation. For example, for the `ROCPROFILER_CALLBACK_TRACING_HIP_RUNTIME_API` and `ROCPROFILER_CALLBACK_TRACING_HIP_COMPILER_API`, the payload must be casted to `rocprofiler_callback_tracing_hip_api_data_t*`, which contains the arguments to the function and the return value when exiting the function. The payload field is a valid pointer only during the invocation of the callback function(s).

- `user_data`: Stores data in between callback phases. This value is unique for every instance of an operation. For example, for a tool library to store the timestamp of the `ROCPROFILER_CALLBACK_PHASE_ENTER` phase for the ensuing `ROCPROFILER_CALLBACK_PHASE_EXIT` callback, the data can be stored using:

  ```
  void
  callback_func(rocprofiler_callback_tracing_record_t record,
                rocprofiler_user_data_t*               user_data,
                void*                                  cb_data)
  {
    auto ts = rocprofiler_timestamp_t{};
    rocprofiler_get_timestamp(&ts);

    if(record.phase == ROCPROFILER_CALLBACK_PHASE_ENTER)
    {
        user_data->value = ts;
    }
    else if(record.phase == ROCPROFILER_CALLBACK_PHASE_EXIT)
    {
        auto delta_ts = (ts - user_data->value);
        // ... etc. ...
    }
    else
    {
        // ... etc. ...
    }
  }
  ```

  The `callback_data` is passed to `rocprofiler_configure_callback_tracing_service` as the value of

`callback_args` to *subscribe to callback tracing services*.

5.3 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
```

5.4 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_REGI`) provides a globally unique identifier for the specific kernel symbol along with the kernel name and several other static properties of the kernel such as scratch size, scalar general purpose register count, and so on.

> **Note**
>
> The kernel identifiers for two identical kernel symbols with the same properties (kernel name, scratch size, and so on) that are part of similar code objects loaded for different GPU agents will still be unique. Furthermore, the identifier for a code object and its kernel symbols after being unloaded and then reloaded, will also be unique.

Here is the general sequence of events when a code object is loaded and unloaded:

1. Callback: load code object
    - kind=ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT
    - operation=ROCPROFILER_CODE_OBJECT_LOAD
    - phase=ROCPROFILER_CALLBACK_PHASE_LOAD
2. Callback: load kernel symbol
    - kind=ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT
    - operation=ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER
    - phase=ROCPROFILER_CALLBACK_PHASE_LOAD
    - Repeats for each kernel symbol in code object
3. Execute application
4. Callback: unload kernel symbol
    - kind=ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT
    - operation=ROCPROFILER_CODE_OBJECT_DEVICE_KERNEL_SYMBOL_REGISTER
    - phase=ROCPROFILER_CALLBACK_PHASE_UNLOAD
    - Repeats for each kernel symbol in code object
5. Callback: unload code object
    - kind=ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT
    - operation=ROCPROFILER_CODE_OBJECT_LOAD
    - phase=ROCPROFILER_CALLBACK_PHASE_UNLOAD

Note

ROCprofiler-SDK doesn't provide an interface to query information outside of the code object tracing service. If you wish to associate kernel names with kernel tracing records, the tool must be configured to create a copy of the relevant information when the code objects and kernel symbol are loaded. However, any constant string fields like `const char* kernel_name` don't need to be copied as these are guaranteed to be valid pointers until after ROCprofiler-SDK finalization. If a tool decides to delete its copy of the data associated with a code object or kernel symbol identifier when the code object and kernel symbols are unloaded, it is highly recommended to flush all buffers that might contain references to that code object or kernel symbol identifier before deleting the associated data.

For a sample of code object tracing, see samples/code_object_tracing.

CHAPTER

SIX

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.

6.1 Definitions

Profile Config: A configuration to specify the counters to be collected on an agent. This must be supplied to various counter collection APIs to initiate collection of counter data. Profiles are agent-specific and can't be used on different agents.

Counter ID: Unique Id (per-architecture) that specifies the counter. The counter Id can be used to fetch counter information such as its name or expression.

Instance ID: Unique record Id that encodes the counter Id and dimension for a collected value.

Dimension: Dimensions help to provide context to the raw counter values by specifying the hardware register that is the source of counter collection such as a shader engine. All counter values have dimension data encoded in their instance Id, which allows you to extract the values for individual dimensions using functions in the counter interface. The following dimensions are supported:

```
    ROCPROFILER_DIMENSION_XCC,             ///< XCC dimension of result
    ROCPROFILER_DIMENSION_AID,             ///< AID dimension of result
    ROCPROFILER_DIMENSION_SHADER_ENGINE,   ///< SE dimension of result
    ROCPROFILER_DIMENSION_AGENT,           ///< Agent dimension
    ROCPROFILER_DIMENSION_SHADER_ARRAY,    ///< Number of shader arrays
    ROCPROFILER_DIMENSION_WGP,             ///< Number of workgroup processors
    ROCPROFILER_DIMENSION_INSTANCE,        ///< From unspecified hardware register
```

6.2 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:

6.2.1 tool_init() setup

Similar to tracing services, you must create a context and a buffer to collect the output when initializing the tool.

> **Note**
>
> `Buffered_callback` in `rocprofiler_create_buffer` is invoked with a vector of collected counter samples, when the buffer is full. For details, see the *Buffered callback* section.

```
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_buffered_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");
```

6.2.1.1 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;

    // 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,
```

(continues on next page)

(continued from previous page)

```
                            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]);
                             }
                             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 version;
        ROCPROFILER_CALL(
            rocprofiler_query_counter_info(
                counter, ROCPROFILER_COUNTER_INFO_VERSION_0, static_cast<void*>(&
↪version)),
            "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_profile_config`.

```
    // Create and return the profile
    rocprofiler_profile_config_id_t profile;
    ROCPROFILER_CALL(rocprofiler_create_profile_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_profile_config` are *agent-specific* and can't be used to construct profiles for other agents.

6.2.2 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_profile_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`.

6.2.3 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_agent_set_profile_callback_t set_config,
            void*)
```

The profile to be used for this agent is specified by calling `set_config(agent, profile)`.

6.2.4 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_record_counter_t*>(header->payload);
            rocprofiler_counter_id_t counter_id = {.handle = 0};
```

(continues on next page)

(continued from previous page)

```
            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;
                rocprofiler_query_record_dimension_position(record->id, dim.id, &pos);
                ss << "{" << dim.name << ": " << pos << "},";
            }
            ss << "] Value [D]: " << record->counter_value << "),";
        }
    }
```

6.3 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.

6.4 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.

6.4.1 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.

6.4.2 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.
```

- `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.

6.5 Kernel Serialization

In *dispatch counting* mode, counter collection requires serialized execution of kernels on a target device to function. Kernel serialization isolates kernel executions, which helps to collect performance counter data. However, kernel serialization also leads to deadlock when applications requiring two kernels to execute on the same device simultaneously (co-dependent kernels) in dispatch counting 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 specifying `filter` tag in the rocprofv3's PMC file.

- Use ROCprofiler-SDK's device-wide counter collection mode to collect performance data. You can use tools such as RDC and PAPI to collect information. Note that the device-wide counter collection captures data for all executions on the device and not specific to the kernels.

CHAPTER

SEVEN

ROCPROFILER-SDK 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.

7.1 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")));
}
```

7.2 Internal implementation of API function

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

7.3 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
```

7.4 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);
}
}
```

7.5 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 an example of dispatch table chaining, see samples/intercept_table.

CHAPTER

EIGHT

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.

Note

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.

CHAPTER

NINE

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.

9.1 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.

9.2 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
{
// saves the data provided to rocprofiler_configure
struct ToolData
{
    uint32_t                                version;
    const char*                             runtime_version;
```

(continues on next page)

(continued from previous page)

```
    uint32_t                                priority;
    rocprofiler_client_id_t                 client_id;
};

// tool initialization function
int
tool_init(rocprofiler_client_finalize_t fini_func,
          void* tool_data_v);

// tool finalization function
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;
}
```

9.3 Tool initialization

> **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 instances. 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.

9.4 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 finalizes the tool after 10 seconds
    auto explicit_finalize = [](rocprofiler_client_finalize_t finalizer,
                                rocprofiler_client_id_t* client_id)
    {
```

(continues on next page)

(continued from previous page)

```
        std::this_thread::sleep_for(std::chrono::seconds{ 10 });
        finalizer(client_id);
    };

    // start the context
    rocprofiler_start_context(ctx);

    // 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.

9.5 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);
```

(continues on next page)

(continued from previous page)

```
    // Save your contexts
    tool_data->contexts.emplace_back(ctx);

    // Associate code object tracing with this context
    rocprofiler_configure_callback_tracing_service(
        ctx,
        ROCPROFILER_CALLBACK_TRACING_CODE_OBJECT,
        nullptr,
        0,
        tool_tracing_callback,
        tool_data);

    // ... Associate services with contexts ...

    return 0;
}

void
tool_fini(void* tool_data);
}

extern "C"
{
rocprofiler_tool_configure_result_t*
rocprofiler_configure(uint32_t                 version,
                      const char*              runtime_version,
                      uint32_t                 priority,
                      rocprofiler_client_id_t* client_id)
{
    // (optional) Provide a name for this tool to rocprofiler
    client_id->name = "ExampleTool";

    // Info provided back to tool_init and tool_fini
    auto* my_tool_data = new rocp_tool_data{ version,
                                             runtime_version,
                                             priority,
                                             client_id,
                                             nullptr };

    // Create configure data
    static auto cfg =
        rocprofiler_tool_configure_result_t{ sizeof(rocprofiler_tool_configure_result_t),
                                             &tool_init,
                                             &tool_fini,
                                             my_tool_data };

    return &cfg;
}
```

CHAPTER

TEN

ROCPROFILER SDK DEVELOPER API

10.1 Topics

10.2 Namespaces

10.2.1 Namespace List

10.2.2 Namespace Members

10.2.2.1 Namespace Members

10.2.2.2 Namespace Members

10.2.2.3 Namespace Members

10.2.2.4 Namespace Members

10.2.2.5 Namespace Members

10.3 Data Structures

10.3.1 Data Structures

10.3.2 Data Structure Index

10.3.3 Class Hierarchy

10.3.4 Data Fields

10.3.4.1 All

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

Data Fields

10.3.4.2 Data Fields - Functions

10.3.4.3 Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

Data Fields - Variables

10.3.4.4 Data Fields - Typedefs

10.4 Files

10.4.1 File List

10.4.2 Globals

10.4.2.1 All

Globals

Globals

Globals

Globals

Globals

Globals

CHAPTER

ELEVEN

COMPARING ROCPROFILER-SDK TO OTHER ROCM PROFILING TOOLS

ROCprofiler-SDK is an improved version of ROCm profiling tools that enables more efficient implementations and better thread safety while avoiding problems that plague the former implementations of ROCProfiler and ROCTracer. Here are the distinct ROCprofiler-SDK features, which also highlight the improvements over ROCProfiler and ROCTracer:

- Improved tool initialization
- Support for simultaneous use of the same services by multiple tools
- Simplified control of one or more data collection services
- Improved error checking and logging
- Backward ABI compatibility
- PC sampling (beta implementation)

The former implementations allow a tool to access any of the services provided by ROCProfiler or ROCTracer, such as API tracing and kernel tracing, by calling `roctracer_init()` when an ROCm runtime is initially loaded. As the calling tool is not required to specify during initialization, the services it needs to use, the libraries must be effectively prepared for any service to be available anytime. This behavior introduces unnecessary overhead and makes thread-safe data management difficult, as tools generally don't use all the available services. For example, ROCTracer always installs wrappers around every runtime API and adds indirection overhead through the ROCTracer library to check for the current service configuration in a thread-safe manner.

ROCprofiler-SDK introduces *context* to solve the preceding issues. Contexts are effectively bundles of service configurations. ROCprofiler-SDK provides a single opportunity for a tool to create as many contexts as required. A tool can group all services into one context, create one context per service, or choose a mix. This change in the design allows ROCprofiler-SDK to be aware of the services that might be requested by a tool at any given time. The design change empowers ROCprofiler-SDK to:

- Avoid unnecessary preparation for services that are never used. If no registered contexts request HSA API tracing, no wrappers need to be generated.
- Perform more extensive checks during service specification and inform a tool about potential issues early.
- Allow multiple tools to use certain services simultaneously.
- Improve thread safety without introducing parallel bottlenecks.
- Manage internal data and allocations more efficiently.

CHAPTER

TWELVE

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 12.1: Comparison of ROCprofiler Command-Line Tool's options

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

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Basic tracing options	Scratch Memory Trace	*Not Available*	*Not Available*	*–scratch-memory-trace*	New option to trace scratch memory operations	
Basic tracing options	Marker Trace(ROCTx)	*–roctx-trace*	*–roctx-trace*	*–marker-trace*	Improved ROCTx library with more features	
Basic tracing options	Memory Copy Trace	Part of HIP and HSA Traces	Part of HIP and HSA Traces	*–memory-copy-trace*	Provides granularity for memory move operations	
Basic tracing options	Kernel Trace	*–kernel-trace*	*–kernel-trace*	*–kernel-trace*	Performance improvement.	
Granular tracing options	HIP runtime trace	Part of *–hip-trace* option	Part of *–hip-trace* option	*–hip-runtime-trace*	For collecting HIP Runtime API Traces, e.g. public HIP API functions starting with 'hip' (i.e. hipSetDevice).	
Granular tracing options	HIP compiler trace	*Not Available*	*Not Available*	*–hip-compiler-trace*	For collecting HIP Compiler generated code Traces, e.g. HIP API functions starting with '__hip' (i.e. __hipRegisterFatBinary).	
Granular tracing options	HSA core API trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-core-trace*	New option for collecting only HSA API Traces (core API), e.g. HSA functions prefixed with only *hsa_* (i.e. hsa_init)	

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Granular tracing options	HSA AMD trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-amd-trace*	For collecting HSA API Traces (AMD-extension API), e.g. HSA function prefixed with *hsa_amd_* (i.e. hsa_amd_coher	
Granular tracing options	HSA Image Extension trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-image-trace*	New option for collecting HSA API Traces (Image-extenson API), e.g. HSA functions prefixed with only *hsa_ext_image_* (i.e. hsa_ext_image_	
Granular tracing options	HSA Finalizer trace	Part of *–hsa-trace* option	Part of *–hsa-trace* option	*–hsa-finalizer-trace*	New option for collecting HSA API Traces (Finalizer-extension API), e.g. HSA functions prefixed with only *hsa_ext_progra* (i.e. hsa_ext_progra	
Aggregate tracing options	Sys Trace	*–sys-trace* [hip-trace|hsa-trace|roctx-trace|kernel-trace]	*–sys-trace* [hip-trace|hsa-trace|roctx-trace|kernel-trace]	` -s, –sys-trace` [hip-trace|hsa-trace|scratch-trace|memory-copy-trace|roctx-trace|kernel-trace]	Extends the sys trace options with more features	

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Aggregate tracing options	Runtime Trace	*Not available*	*Not available*	` -r, –runtime-trace` [hip-runtime-trace|scratch-trace|memory-copy-trace|roctx-trace|kernel-trace]	New option to aggregate trace operations	
Kernel naming options	Kernel Name Mangling	*Not Available*	*Not Available*	*-M, –mangled-kernels*	New option for mangled kernel names	
Kernel naming options	Kernel Name Truncation	*–basenames <on|off>*	*–basenames*	*-T, –truncate-kernels*	New option for truncating the demangled kernel names	
Kernel naming options	Kernel Rename	*–roctx-rename*	*Not available*	*–kernel-rename*	New option to use region names defined by roctxRangePush/roctxRang regions to rename the kernels	
Post-processing tracing options	Statistics	–stats	*Not Available*	–stats	Statistics for the collected traces	
Post-processing tracing options	Summary	*Not available*	*Not available*	*-S, –summary*	New option to output a single summary of tracing data after the profiling session	*rocprof* generated the post-processing step's summary, stats, JSON, and database files with much less information.
Post-processing tracing options	Summary Per Domain	*Not available*	*Not available*	*-D, –summary-per-domain*	New option to output summary for each tracing domain after the profiling session	*rocprof –stats* option had less number of domains in the summary reports than *rocprofv3*

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Post-processing tracing options	Summary Groups	*Not available*	*Not available*	*–summary-groups REGULAR_EXPRESS*	New option to output a summary for each set of domains matching the regular expression, e.g. 'KERNEL_DISPAT(will generate a summary from all the tracing data in the KERNEL_DISPAT(and MEMORY_COPY domains	COPY'
Summary options	Summary Output File	*Not available*	*Not available*	*–summary-output-file SUMMARY_OUTPU*	New option to output summary to a file, stdout, or stderr (default: stderr)	
Summary options	Summary Units	*Not available*	*Not available*	*-u , –summary-units*	New option to output summary in desired time units {sec,msec,usec	
Display options	List Metrics	*–list-basic*, *–list-derived*	*–list-counters*	*-L*, *–list-metrics*	A valid YAML is supported for this option now	
Perfetto-specific options	Perfetto data collection backend	*Not available*	*Not available*	*–perfetto-backend* {inprocess,system}	New option for perfetto data collection backend. 'system' mode requires starting traced and perfetto daemons	*rocprofv2* used only in-process collection for perfetto plugin, However, *rocprofv3* give the option to the user

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Perfetto-specific options	Perfetto Buffer Size	*Not available*	Setting env variable *rocprofiler_PERFETT* to the desired buffer size	*–perfetto-buffer-size* {KB}	New option to define size of buffer for perfetto output in KB. default: 1 GB	
Perfetto-specific options	Perfetto Buffer fill Policy	*Not available*	*Not available*	*–perfetto-buffer-fill-policy* {discard,ring_buffe	New option or handling new records when perfetto has reached the buffer limit	*rocprofv2* always used *TraceConfig_BufferConfig_FillPolicy_RIN* fill policy.
Perfetto-specific options	Perfetto shared memory size	*Not available*	*Not available*	*–perfetto-shmem-size-hint* KB	New option to define perfetto shared memory size hint in KB. default: 64 KB	
Filtering options	Kernel Filtration options for Counter Collection	Supported in input.xml file (supports range, gpu and kernel filtration)	kernel: <kernel_name> (can only be provided in input.txt file)	*–kernel-include-regex*, *–kernel-exclude-regex*, *–kernel-iteration-range*	Extensive control over output options using regular expressions	
I/O options	Output Directory	*-d* <data directory>	*-d* | *–output-directory*	*-d* OUTPUT_DIRECT *–output-directory* OUTPUT_DIRECT	rocprofv3 supports special keys for runtime values, e.g. %pid% gets replaced by the process ID	
I/O options	Output File	*-o* <output file>	*-o* | *–output-file-name*	*-o* OUTPUT_FILE, *–output-file* OUTPUT_FILE	rocprofv3 supports special keys for runtime values, e.g. %pid% gets replaced by the process ID	

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
I/O options	Logging	Minimal logging via environment variable	Minimal logging via environment variable	–log-level {fatal,error,warnin	Extensive logging options	
I/O options	Plugins	*Not Available*	plugin support for different output formats	Replaced by *–output-format* option	Not needed as rocprofv3 supports multiple output formats	
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.

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
I/O options	Counter Collection	Supports input text and XML format	Only supports input text format	Input support for text, YAML and JSON formats	# Its 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	
I/O options	Providing Custom metrics file	*-m* <metric file>	*-m* <metric file>	Not available	Not yet in rocprofv3	
Advanced options	Preload	*Not Available*	*Not Available*	–preload	Libraries to prepend to LD_PRELOAD (usually for sanitizers)	
Trace Control options	Trace Period	*–trace-period*	*-tp* | *–trace-period*	*Not available*	Not yet in rocprofv3	
Trace Control options	Trace start	*–trace-start* <*on*|*off*>	*Not available*	*Not available*	Not yet in rocprofv3	
Trace Control options	Flush Interval	*–flush-rate*	*–flush-interval*	*Not available*	Not applicable for rocprofv3	
Trace Control options	Merge Traces	*–merge-traces*	*Not available*	*Not available*	Not yet in rocprofv3	
Legacy options	Timestamp On/Off	*–timestamp* <*on*|*off*>	*Not available*	*Not available*	Not applicable for rocprofv3	
Legacy options	Context wait	*–ctx-wait*	*Not available*	*Not available*	Not applicable for rocprofv3	

continues on next page

Table 12.1 – continued from previous page

Category	Feature	rocprof	rocprofv2	rocprofv3	Improvements	Notes
Legacy options	Context Limit	*–ctx-limit <max number>*	*Not available*	*Not available*	Not applicable for rocprofv3	
Legacy options	Code Object Tracking	*–obj-tracking <on|off>*	Always `ON` in rocprofv2	Always `ON` in rocprofv3		
Legacy options	Heartbeat	*–heartbeat <rate sec>*	*Not available*	*Not available*	Not applicable for rocprofv3	

CHAPTER

THIRTEEN

TIMING DIFFERENCE BETWEEN ROCPROFV3 AND ROCPROFV1/V2

`rocprofv3` has improved the accuracy of timing information by reducing the tool overhead required to collect data and reducing the interference to the timing of the kernel being measured. The result of this work is a reduction in variance of kernel times received for the same kernel execution and more accurate timing in general. These changes have not been backported (and will not be backported) to rocprofv1/v2, so there can be substantial (20%) differences in execution time reported by v1/v2 vs v3 for a single kernel execution. Over a large number of samples of the same kernel, the difference in average execution time is in the low single digit percentage time with a much tighter variance of results on rocprofv3. We have included testing in the test suite to verify the timing information outputted by rocprofv3 to ensure that the values we are returning are accurate.

CHAPTER

FOURTEEN

DEFAULT RUN OF ROCPROFV3 AND ROCPROFV1/V2

`rocprofv3` has a different default behavior than rocprofv1/v2 when being run without any option. The default behavior of rocprofv3 is to collect all available agents on the system and to output it in `csv` format. The default behavior of rocprofv1/v2 was to output the *kernel traces* in CSV format. In rocprofv3, kernel traces can be obtained by using `--kernel-trace` option.

CHAPTER

FIFTEEN

LICENSE