rocprofiler Documentation

Release 2.0.0

Advanced Micro Devices, Inc.

Sep 27, 2024

CONTENTS

ROCProfiler is a powerful tool for profiling HIP and ROCm applications on AMD ROCm software. Profiling is used to identify performance bottlenecks in applications and to optimize their performance. ROCProfiler offers a library for developing applications with profiling features and a command-line interface (CLI) tool for profiling compiled applications. The library and CLI tool for the first version of ROCProfiler are named ROCProfiler and `rocprof` respectively, whereas, the library and the CLI tool for the second version of ROCProfiler are ROCProfilerV2 and `rocprofv2`. This documentation covers information on both the ROCProfiler versions. For more information on ROCprofiler, refer to *What is ROCProfiler?*

You can access ROCProfiler on https://github.com/ROCm/rocprofiler.

Install

- *Installing ROCProfiler*

The documentation is structured as follows:

Tutorials

- MatrixTranspose application tutorial
- ROCm examples
- HIP examples

How to

- ROCProfiler CLI
  - *Using rocprof*
  - *rocprof command help*
- ROCProfilerV2 CLI
  - *Using rocprofv2*
  - *Using rocsys*
  - *rocprofv2 command help*

References

- ROCProfiler
  - *ROCProfiler library specification*
  - *ROCProfiler API library*
    - *Modules*
    - *Data Structures*
    - *Files*
- ROCProfilerV2
  - *ROCProfilerV2 API*

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

You can find licensing information on the Licensing page.

CHAPTER

ONE

WHAT IS ROCPROFILER?

ROCProfiler is a powerful tool for profiling HIP and ROCm applications on AMD ROCm software. Profiling is used to identify performance bottlenecks in applications and to optimize their performance. ROCProfiler provides command-line tools for profiling precompiled applications. The ROCProfiler tool is implemented using the ROCProfiler and ROCTracer libraries, and provides two primary features to profile GPU-based applications or kernels:

- Application tracing: This basic feature of the ROCProfiler is used to trace the execution of an application, with timestamps for the start and end of each API call, and kernel execution.
- Performance counter collection: This feature collects performance counters for each API call and kernel execution.

The first version of ROCProfiler is named ROCProfiler while the second version is ROCProfilerV2. The two versions are similar in terms of basic functionalities offered, which are, application tracing and kernel profiling. However, there are differences between the ROCProfiler command-line tool *rocprof* and the ROCProfilerV2 command-line tool *rocprofv2* in terms of additional functionalities, default outputs supported by the tools, and supported AMDGPUs.

To demonstrate the usage of both versions of ROCProfiler with various options, this document refers to the Matrix-Transpose application as an example.

CHAPTER

TWO

INSTALLING ROCPROFILER

Since the ROCProfiler library, `rocprof`, and `rocprofv2` are included as standard components of the ROCm distribution, the simplest method of installation is to perform a full installation of ROCm as described in Installation Guide. This approach is straightforward and dependable, making it suitable for most users.

Alternatively, installing ROCProfiler from the source files offers some advantages. This option is only recommended for experienced users when necessary due to the additional complexity.

2.1 Installing from source files

The following section discusses how to install and build from the source files.

2.1.1 Prerequisites

Installing ROCProfiler from the source requires:

- AMD GPU driver for supported GPUs
- Linux system supported by ROCm as described in System requirements
- ROCm installed in the system
- Packages necessary for ROCProfiler on supported Linux distributions as described in the following:

Ubuntu 20.04 and 22.04

```
sudo apt install libudev-dev libnuma1 libnuma-dev Python3 python3-pip gcc g++ make cmake␣
↪doxygen
```

RHEL 8.6 and 9.2/CentOS 7

- RHEL 8.6 and 9.2:

```
yum install -y systemd-devel numactl numactl-devel Python3 python3-pip gcc gcc-
↪g++ make cmake libatomic doxygen
```

- CentOS 7:

```
yum install -y systemd-devel numactl numactl-devel Python3 python3-pip gcc gcc-
↪g++ make cmake libatomic doxygen
```

SLES 15.3/SLES Tumbleweed

- SLES 15.3:

```
zypper in libudev-devel libnuma1 libnuma-devel Python3 python3-pip gcc gcc-g++␣
↪make cmake libatomic doxygen
```

- SLES Tumbleweed:

```
zypper install systemd-devel libnuma1 libnuma-devel Python3 python3-pip gcc␣
↪gcc-g++ make cmake libatomic doxygen
```

- Python libraries such as `CppHeaderParser`, `argparse`, and `sqlite3`. To install the libraries based on the default Python installation on the system, use:

```
pip3 install cppheaderparser argparse sqlite3
```

2.1.2 Building from source

With the dependencies installed, you can now build `rocprofiler` from the source code using the steps given below:

1. After cloning the repository, you can use `CMake` to compile and install `rocprofiler`. To use `CMake`, set the environment variable:

```
export CMAKE_PREFIX_PATH=/opt/rocm/include/hsa:/opt/rocm
```

2. Then, build `rocprofiler` with a premade bash script.

```
./build.sh
```

3. You now have two options for utilizing the newly built `rocprofiler` and its associated libraries.

   a. The first option involves setting an environment variable to indicate the location of the executable and libraries. This enables the system to use the new `rocprof` and related libraries instead of the default version.

   ```
   export LD_LIBRARY_PATH=<BUILD_LOCATION>:$LD_LIBRARY_PATH
   export PATH=<BUILD_LOCATION>/bin:$PATH
   ```

   b. A second option is to specify the desired location for installation of the executable and libraries if you do not want the installation process to move the executable and libraries to the default location. To specify the installation location, set environment variable:

   ```
   export CMAKE_INSTALL_PREFIX=<LOCAL INSTALL FOLDER>
   ```

4. Then, to install `rocprof` in the build directory, run:

```
cd build
make install
```

> **Note:** The Makefile is generated by `CMake`.

CHAPTER

THREE

USING ROCPROF

`rocprof` is a powerful tool for profiling HIP applications on AMD ROCm platforms. It can be used to identify performance bottlenecks in applications and to optimize their performance. `rocprof` provides a variety of profiling data, including performance counters, hardware traces, and runtime API/activity traces. This document provides a detailed description of the features and usage of the `rocprof` command-line tool with a focus on these two primary features:

- Application Tracing: This basic feature of `rocprof` is used to trace the execution of an application, with start/end timestamps for each API call and kernel execution. [Read more here](#application-tracing).
- Performance Counter Collection: This powerful feature collects performance counters for each API call and kernel execution. [Read more here](#performance-counter-collection).

To see all the `rocprof` options, refer to *rocprof command help*, or run the following from the command line:

```
rocprof --help
```

3.1 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 which kernel took the longest time to execute, and what percentage of time was spent on memory copy.

There are two ways to use application tracing: `rocprof` and `ROCTracer` API.

- `rocprof` is a command line interface (CLI) profiler that can be used on the applications running on ROCm-supported GPUs, without the requirement of any code modification in the application.
- `ROCTracer` API is a library that requires minor code modification in the application to be traced but provides greater flexibility, such as adjusting execution based on profiling results.

3.1.1 Command-line options for application tracing

The `rocprof` CLI allows you to trace the entire execution of *HIP* applications. It allows tracing at different levels such as *HIP*-level, *HSA*-level, and system-level. These levels can be selected by supplying the respective command-line options to *rocprof*.

The command-line options used with `rocprof` for tracing:

Options	Description
`-d <output directory>`	To specify the directory where the profiler stores traces and the profiling data. The profiler stores the profiling data in a temporary directory [/tmp] by default, which is removed automatically after a specific period. Specifying a directory explicitly allows you to prevent loss of data.
`--hip-trace`	To trace API execution stats, *HIP* API calls, and copy operation calls.|br| For more information refer to *HIP Trace*.
`--hsa-trace`	To trace API execution stats, *HIP* API calls, and copy operation calls.|br| For more information refer to *HSA Trace*.
`--roctx-trace`	To enable `roctx` application code annotation trace. Allows you to trace a particular block of code when Markers and Ranges are specified in the application code. For more information refer to *ROCTx Trace*.
`--stats`	To trace API execution stats and kernel execution stats. For more information refer to stats-file.
`--sys-trace`	To trace API execution stats, *HIP* and *HSA* API calls, and copy operation calls. For more information refer to *System Trace*.

3.1.2 HIP Trace

Use the `--hip-trace` option to collect execution trace data for the entire application with `rocprof`, including `HIP` API functions and their asynchronous activities at the runtime level.

Usage:

```
rocprof -d outputFolder --hip-trace ./Matrixtranspose
```

The above command generates three groups of files: viewable trace data, statistics files, and intermediate tracing data.

Note: This document does not discuss the intermediate tracing data as it is not designed to be read by users.

Viewable Trace Data

The viewable trace data is available in `results.json`, which is a `JSON` file that follows the Chromium Project's trace-event format. You can view the trace using viewing tools such as *Chrome Tracing <chrome://tracing/>* or Perfetto UI. In the following figure a short segment of the trace data is viewed:

Fig. 3.1: Viewing HIP Trace

In the time axis at the top of the following figure there is a small, highlighted region between 0.22s to 0.24s, that indicates the currently selected time range.

Fig. 3.2: Time Range

Below the time axis in the following figure there are Gantt chart-style boxes that show the duration of each task. There are three rows of tasks divided by the black rows mentioning their categories. The first row is the *CPU HIP API*, which lists the execution time of each API trace.

Fig. 3.3: Duration of API Trace Execution

In the following figure the second row titled *COPY* shows the tasks completed by the copy engine. See that the `CopyHostToDevice` and `CopyDeviceToHost` tasks are being completed at the beginning and the end of the time range, respectively.

Fig. 3.4: Copy Tasks

In the following figure the GPU tasks are listed at the bottom. Note that the `matrixTranspose` kernel is executed from about 0.185s to 0.20s.

Fig. 3.5: GPU Tasks

Statistics Files

The statistics files include:

- `results.stats.csv` for kernel statistics
- `results.hip_stats.csv` for `HIP` API
- `results.copy_stats.csv` for activity statistics

The files are organized in comma-separated values (CSV) format, with the columns:

- **Name:** Name of the action
- **Calls:** Number of invocations
- **TotalDurationNS:** Total duration in nanosecond
- **AverageNS:** Average time in nanoseconds required to execute the action
- **Percentage:** Percentage of the action with respect to the complete execution of the application

3.1.3 HSA Trace

The `HIP` runtime library is implemented with the low-level `HSA` runtime. To trace the application at a lower level, you can use `rocprof` to collect application traces at the `HSA` runtime level. In general, tracing at the `HIP`-level is recommended for most users. You are advised to use `HSA` trace only if you are familiar with `HSA` runtime.

`HSA` trace contains the start/end time of `HSA` runtime API calls and their asynchronous activities. Use the `--hsa-trace` option with rocprof to collect `HSA`-level trace:

```
rocprof --hsa-trace ./MatrixTranspose
```

As in `HIP` trace, the generated `HSA` trace also includes three groups of files, namely viewable traces, statistics files, and intermediate tracing data. You can visualize the generated `results.json` using third-party tools such as Perfetto, as shown in the following figure.

Fig. 3.6: Viewing HSA Trace

In the preceding figure you can see that `HSA` trace shows `HSA` trace rows, just like `HIP` trace shows the `HIP` API row in *{numref}hip-trace-visualize*. However, note that there are more `HSA` API calls as compared to `HIP` API calls because `HSA` works closer to the hardware.

Note: GPU ID in the *HSA* visualization is always 0.

The statistics files are:

- `results.stats.csv` for kernel statistics
- `results.hsa_stats.csv` for *HSA* API
- `results.copy_stats.csv` for memory copy statistics

Each file is a CSV table with columns as described in the stats-file.

3.1.4 System Trace

The `rocprof` tool can also generate both the `HIP` and `HSA` traces together with the `--sys-trace` option.

```
rocprof --sys-trace ./MatrixTranspose
```

The following figure shows the generated `results.json` visualized using Perfetto. It contains sections from both `HIP` and `HSA` trace.

Fig. 3.7: Viewing Sys Trace

3.1.5 ROCTx Trace

In certain situations, such as debugging performance issues in large-scale GPU programs, API-level tracing may be too fine-grained to provide a big picture of the program execution. In such cases, you might find it 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 `roctx`. 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 fine grain the scope specified within a range using further nested ranges. The `rocprof` tool 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 can help you see when a line of code is executed.
- `roctxRangeStart`: Starts a range. Ranges can be started by different threads.
- `roctxRangePush`: Starts a new nested range.
- `roctxRangePop`: Stops the current nested range.
- `roctxRangeStop`: Stops the given range.

See `roctx` code annotations in the `MatrixTranspose` application below:

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

// 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);
```

(continues on next page)

(continued from previous page)

```
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);
```

Tip: A version of the MatrixTranspose application instrumented using the ROCTx API is available in the rocprofiler/tests-v2 folder on GitHub.

Using `rocprof` with `roctx-trace` option:

```
rocprof -d outputFolder --roctx-trace ./
```

You can visualize the generated output file `results.json` using Perfetto as shown in the following figure. The sections *Markers* and *Ranges* show the marked events and ranges.

Fig. 3.8: Viewing Roctx Trace

3.2 Tracing Control

The `rocprof` tool provides these customization options:

3.2.1 Filter Tasks

To filter tasks, specify the trace category and the tasks to be traced in an input file.

```
cat input.txt
hsa : hsa_queue_create hsa_amd_memory_pool_allocate
```

Then supply this input file to `rocprof`.

```
rocprof -i input.txt --hsa-trace ./MatrixTranspose
```

The above sample input file generates `HSA` tracing information for only the two events specified in the file.

3.2.2 Adjust Trace Flush Rate

Use the `--flush-rate` option to specify the flush rate in seconds, milliseconds, or microseconds. This determines how often the trace data is dumped to files. In the following example the flush rate is set to 10 microseconds:

```
rocprof --flush-rate 10us --hsa-trace ./MatrixTranspose
```

3.2.3 Function Name Truncation

To truncate the kernel full function name in the trace files to the base name of the function, use `--basename` as shown in the following example:

```
$ rocprof --basename on --hip-trace ./MatrixTranspose

"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"vectoradd_float",1,46373,46373,100.0
```

When `--basename` is not explicitly enabled, the full kernel function name is displayed in the trace:

```
$ rocprof --basename off --hip-trace ./MatrixTranspose

"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"vectoradd_float(float*, float const*, float const*, int, int)",1,45633,45633,100.0
```

3.2.4 Tracing Control for API or Code Block

To enable selective tracing for a `HIP` API or code block instead of the entire application, follow these steps:

1. Enclose the API or code block within *roctracer_start()* and *roctracer_stop()*. This ensures that tracing starts only when it encounters *roctracer_start()* and stops once it encounters *roctracer_stop()*. See the usage of these API calls here, where the user wants to trace only *hipMemcpy()* API:

```
#include <roctracer/include/roctracer_ext.h>

// allocate the memory on the device side
hipMalloc((void**)&gpuMatrix, NUM * sizeof(float));
hipMalloc((void**)&gpuTransposeMatrix, NUM * sizeof(float));

roctracer_start();

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

roctracer_stop();
```

2. Use *–trace-start* off to disable application tracing from the beginning and start tracing only when *roctracer_start()* is encountered.

```
$ rocprof --trace-start off --hip-trace MatrixTranspose

cat results.hip_stats.csv
```

(continues on next page)

(continued from previous page)

```
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"hipMemcpy",10,255048886,25504888,100.0
```

3.2.5 Set Initial Delay, Periodic Sample Length and Rate

Use option *-trace-period <delay:length:rate>* to fine tune tracing by setting the following attributes:

- **Initial Delay:** The time interval between the start of the profiler and start of tracing. So an initial delay of 10 ms causes the tracing to commence after 10 ms since the start of the profiler.
- **Periodic Sample Length:** The duration for which tracing runs
- **Rate:** Rate at which the tracing results are flushed to the user

Example: Tracing with a delay of 10ms, length of 1ms and rate of 10ms

```
$ rocprof --hip-trace --trace-period 10ms:1ms:10ms MatrixTranspose

cat results.hip_stats.csv
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"hipMemcpy",6,6358906,1059817,100.0
```

Example: Tracing with a delay of 10ms, length of 1ms and rate of 1ms

```
$ rocprof --hip-trace --trace-period 10ms:1ms:1ms MatrixTranspose

cat results.hip_stats.csv
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"hipMemcpy",11,272473856,24770350,99.9468484848238
"hipMalloc",2,102871,51435,0.037734380837192355
"hipFree",2,39940,19970,0.014650495967157534
"hipGetDeviceProperties",1,2090,2090,0.0007666383718417439
```

3.3 ROCTracer API

The *ROCTracer* APIs are runtime-independent APIs for tracing runtime calls and asynchronous activity, like GPU kernel dispatches and memory moves. The tracing includes callback API for runtime API tracing and activity API for asynchronous activity records logging. You can utilize these APIs to develop a tracing tool or to implement tracing within an application. Refer to the ROCTracer API Specification for more information.

To use the *ROCTracer* API, link the application with *ROCTracer* using the API header and dynamic library as shown below:

- **API header:** */opt/rocm-{version}/include/roctracer/roctracer.h*
- **Dynamic library (.so):** */opt/rocm-{version}/lib/libroctracer64.so.<version major>*

3.4 Performance Counter Collection

As discussed in the sections above, the application trace mode is limited to providing an overview of program execution and does not provide an insight into kernel execution. To address performance issues, the counter and metric collection functionality of `rocprof` can be used to report hardware component performance metrics during kernel execution.

Counter and metric collection is supported on the following GPUs:

- AMD Radeon Instinct MI25, MI50, MI100, MI2XX
- AMD Radeon VII, Radeon Pro VII

3.4.1 Command-line options for counter collection

The command-line options used with `rocprof` for profiling:

Options	Description
`--list-basic`	To print the list of basic hardware counters. For more information refer to *Listing Performance Counters*.
`--list-derived`	To print the list of derived metrics with formulas.
`-i <.txt/.xml file>`	To retrieve the values of the desired list of basic counters and/or derived metrics. For more information refer to *Input File*.
`-m <.xml file>`	To define new derived metrics or modify the existing metrics defined in the *metrics.xml* by default. For more information refer to *Metric File*.
`-o <output file.csv>`	Specify a name for the output file generated when used with `-i`. For more information refer to *Generated Output*.
`--timestamp <on/off>`	To enable or disable the kernel dispatch timestamps in nanoseconds for events such as dispatch, begin, end, and complete, as described in the following. Default value: `on`. For more information refer to *Generated Output*. * `DispatchNs`: indicates the time when the GPU receives notification to work on a specific kernel as the kernel Architected Queuing Language (AQL) dispatch packet is submitted to the queue. * `BeginNs`: indicates when the kernel begins execution * `EndNs`: time when the kernel finishes execution * `CompleteNs`: inidcates the time when the system receives notification from the GPU about completion of work by the kernel through the completion signal of the AQL dispatch packet.

3.4.2 Listing Performance Counters

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

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

To obtain the list of supported basic counters, use:

```
rocprof --list-basic
```

The derived metrics are calculated from the basic counters using mathematical expressions. To list the supported derived metrics along with their mathematical expressions, use:

```
rocprof --list-derived
```

You can also customize the derived metrics as explained in *Metric File*.

Note: The output generated from the `--list_basic` and `--list_derived` commands can be significant, and is sometimes worth capturing by redirecting the output to a file.

```
rocprof --list-derived ./MatrixTranspose > output.txt
```

3.5 Using `rocprof` for Application Profiling

To profile kernels in GPU applications, define the profiling scope in an input file and use:

```
rocprof -i input.txt ./MatrixTranspose
```

Note: Refer to the MatrixTranspose application tutorial for the example application.

3.5.1 Input File

As mentioned above, an input file is a text file that can be supplied to `rocprof` for basic counter and derived metric collection. It typically consists of four parts: the counters and derived metrics to collect, GPUs to profile, names and range of kernels to profile.

The collected data is written to an output CSV file that has the same name as the input file specified. For example`` -i input.txt`` results in `input.csv` being generated.

Sample Input File

```
# Perf counters group 1
pmc: MemUnitStalled,TCC_MISS[0]
# Filter by dispatches range, GPU index and kernel names
# supported range formats: "3:9", "3:", "3"
range: 0:1
gpu: 0
kernel: matrixTranspose
```

The fields in the input file are as follows:

PMC: The rows in the text file beginning with `pmc:` are the group of basic counters or derived metrics you are interested in collecting. The performance counters can be selected from the output generated by `--list-basic` or `--list-derived` command.

The number of basic counters or derived metrics that can be collected in one run of profiling is limited by the GPU hardware resources. If too many counters/metrics are selected, the kernels need to be executed multiple times to collect the counters/metrics. For multi-pass execution, include multiple rows of `pmc:` in the input file. Counters or metrics in each `pmc:` row can be collected in each run of the kernel.

Note: `rocprof` will provide suggestions to group the counters/metrics if you exceed the hardware limits. You can use this suggestion to split the counters/metrics into group sets and successfully perform counter/metric collection.

Example: to see the suggestion to group the counters/metrics:

```
$ cat input.txt
pmc : Wavefronts, VALUInsts, SALUInsts, SFetchInsts,
FlatVMemInsts,
LDSInsts, FlatLDSInsts, GDSInsts, VALUUtilization, FetchSize,
WriteSize, L2CacheHit, VWriteInsts, GPUBusy, VALUBusy, SALUBusy,
MemUnitStalled, WriteUnitStalled, LDSBankConflict, MemUnitBusy
range: 0 : 1
gpu: 0
kernel:matrixTranspose

$ ropcprof -i input.txt ./MatrixTranspose
Input metrics out of hardware limit. Proposed metrics group set:
group1: FetchSize WriteSize VWriteInsts MemUnitStalled MemUnitBusy
FlatVMemInsts LDSInsts VALUInsts SALUInsts SFetchInsts
FlatLDSInsts GPUBusy Wavefronts
group2: WriteUnitStalled L2CacheHit GDSInsts VALUUtilization
VALUBusy SALUBusy LDSBankConflict
```

Note: The results reported vary depending on the GPU device being profiled.

GPU: The row beginning with the keyword `gpu:` specifies the GPU(s) on which the hardware counters are to be collected. This enables the support for profiling multiple GPUs. You can specify multiple GPUs separated by comma such as `gpu: 1,3`.

Kernel: specifies the names of kernels to profile.

Range: specifies the range of kernel dispatches. Specifying range is helpful in cases where the application causes multiple kernel dispatches and users want to filter some kernel dispatches. In the above example, the range 0:1 depicts that one kernel is profiled.

3.5.2 Metric File

The derived metrics are defined in the `/opt/rocm/lib/rocprofiler/metrics.xml` by default.

Here is an entry from `metrics.xml`:

```
<global>
# Wavefronts
<metric
    name="Wavefronts"
    descr="Total wavefronts."
    expr=SQ_WAVES
></metric>
</global>
```

To override the properties (description/expression) of the derived metrics that are predefined in the `metrics.xml`, redefine the derived metrics in the custom derived metrics file as shown here:

```
#include "gfx_metrics.xml"

<global>
<metric
    name="Wavefronts"
    descr="Total wavefronts. Description redefined by user."
    expr=SQ_WAVES
></metric>
</global>
```

Note that while specifying your custom metrics file, you must include `rocprofiler/test/tool/gfx_metrics.xml` file as all the basic counters are defined here. The basic counters are used in calculating the derived metrics.

Here is an entry from `gfx_metrics.xml`:

```
<gfx9>
<metric name="SQ_WAVES" block=SQ event=4 descr="Count number of waves sent to SQs. (per-
↪simd, emulated, global)"></metric>
<gfx9>
```

You can also define new derived metrics in the custom metrics file as shown here:

```
#include "gfx_metrics.xml"

<gfx9_expr>
<metric name="TotalWorkItems" expr=SQ_WAVES*4  descr="Total number of waves sent to␣
↪SQs(For all simd's). Defined by user." ></metric>
</gfx9_expr>
```

To see the list of customized derived metrics, use:

```
$ rocprof -m custom_metrics.xml --list-derived
gpu-agent1 : TotalWorkItems : Total number of waves sent to SQs(For all simd's). Defined␣
↪by user.
TotalWorkItems = SQ_WAVES*4
```

To collect the values of custom derived metrics, use:

```
cat input.txt
pmc: TotalWorkItems

$ rocprof -i input.txt -m custom_metrics.xml ./MatrixTranspose
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,arch_vgpr,accum_
↪vgpr,sgpr,wave_size,sig,obj,TotalWorkItems
0,"matrixTranspose(float*, float*, int) [clone .kd]",1,0,0,2746,2746,1048576,16,0,0,8,0,
↪16,64,0x0,0x7fead980e800,262144.0000000000
```

Note: You must use the input file to supply the list of derived metrics to be collected while the custom metrics file provides the expressions for calculating those derived metrics. When collecting the custom derived metrics, make sure to mention only those derived metrics in the input file that you have defined in the custom derived metrics file. When not using the option `-m`, `rocprof` refers to the default `metrics.xml` for expressions of all the derived metrics specified in the input file.

The basic counters and derived metrics specific to the AMD GPUs are listed in the xml files under the respective GPU family. The LLVM target `gfx9` corresponds to `MI50` and `MI100` while `gfx90a` corresponds to the `MI200` family. The counters and metrics applicable to all GPUs are listed under global. See the supported GPUs and their respective LLVM targets in the Linux Supported GPUs.

3.5.3 Generated Output

Executing `rocprof` with an input file `input.txt` produces an output CSV (the file name can be specified with the `-o` option) with the counter information as shown below:

Example: Executing `rocprof` on `MatrixTranspose` application with sample input file

```
$ rocprof -i input.txt ./MatrixTranspose
$ cat input.csv
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,
scr,vgpr,sgpr,fbar,sig,obj,MemUnitStalled,TCC_MISS[0]
0,"matrixTranspose(float*, float*, int) [clone .kd]",0,0,0,2614,2614,
1048576,16,0,0,8,24,0,0x0,0x7fbfcb37c580,6.6756117852,4096.
```

Each row of the CSV file is an instance of kernel execution. The columns in the output file are:

- **Index** - kernels dispatch order index
- **KernelName** - kernel name
- **gpu-id** - GPU ID the kernel was submitted to
- **queue-id** - ROCm queue unique ID the kernel was submitted to
- **queue-index** - ROCm queue write index for the submitted AQL packet
- **pid** - system application process ID
- **tid** - system application thread id that submitted the kernel
- **grd** - kernel's grid size
- **wgr** - kernel's work group size
- **lds** - kernel's LDS memory size
- **scr** - kernel's scratch memory size
- **vgpr** - kernel's VGPR size
- **sgpr** - kernel's SGPR size
- **fbar** - kernel's barriers limitation
- **sig** - kernel's completion signal

Example: To enable the timestamp

To enable timestamp in the output, use option `--timestamp on` as shown in the following example.

```
$ rocprof -i input.txt --timestamp on ./MatrixTranspose
$ cat input.csv
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,
wgr,lds,scr,vgpr,sgpr,fbar,sig,obj,
MemUnitStalled,TCC_MISS[0],DispatchNs,BeginNs,EndNs,CompleteNs
0,"matrixTranspose(float*, float*, int) [clone .kd]",0,0,0,2837,2837,
1048576,16,0,0,8,24,0,0x0,
```

(continues on next page)

(continued from previous page)

```
0x7fcd75984580,5.9792124305,4096,87851328156768,
87851334047658,87851334141098,87851334732528
```

Note: The `--timestamp` is on by default.

Example: To specify the output file name.

The default output CSV file has the same name as the input file specified. To specify a name for the output file, use option `-o <file name>` as shown in the following example.

```
$ rocprof -i input.txt --timestamp on -o output.csv ./MatrixTranspose
$ cat output.csv
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,
grd,wgr,lds,scr,vgpr,sgpr,fbar,
sig,obj,MemUnitStalled,
TCC_MISS[0],DispatchNs,BeginNs,EndNs,CompleteNs
0,"matrixTranspose(float*, float*, int) [clone .kd]",0,0,0,
215,215,1048576,16,0,0,8,24,0,0x0,0x7f961080a580,7.0675214726,4096,
91063585321414,91063591158627,91063591252551,91063592018031
```

3.5.4 Profiling Multiple MPI Ranks

To profile multiple MPI ranks use the following command:

```
mpirun ... <mpi args> ... rocprof ... <rocprof args> ... application ... <application␣
↪args>
```

Important: When using *OpenMPI*, you must run `rocprof` inside the `mpirun` command, as shown above, to enable ROCProfiler to handle process forking and launching via `mpirun` and related executables. If you run the `rocprof` command before `mpirun`, then the tool fails with the error *roctracer: Loading 'libamdhip64.so' failed, (null)*.

This execution mode requires the following:

1. Generation of trace data per *MPI* (or process) rank

   ROCm provides a simple bash wrapper that demonstrates how to generate a unique output directory per process as given below:

   ```
   $ cat wrapper.sh
   #! /usr/bin/env bash
   if [[ -n ${OMPI_COMM_WORLD_RANK+z} ]]; then
       # mpich
       export MPI_RANK=${OMPI_COMM_WORLD_RANK}

   elif [[ -n ${MV2_COMM_WORLD_RANK+z} ]]; then
       # ompi
       export MPI_RANK=${MV2_COMM_WORLD_RANK}
   fi

   args="$*"
   ```

(continues on next page)

(continued from previous page)

```
pid="$$"
outdir="rank_${pid}_${MPI_RANK}"
outfile="results_${pid}_${MPI_RANK}.csv"
eval "rocprof -d ${outdir} -o ${outdir}/${outfile} $*"
```

This script:

- Determines the global *MPI* rank (implemented here for *OpenMPI* and *MPICH* only)
- Determines the process id of the *MPI* rank
- Generates a unique output directory using the two

To invoke this wrapper, use the following command:

```
mpirun <mpi args> ./wrapper.sh --hip-trace <application> <args>
```

This generates an output directory for each MPI rank used.

Example:

```
$ ls -ld rank_* | awk {'print $5" "$9'}
4096 rank_513555_0
4096 rank_513556_1
```

2. Combining traces from multiple processes

   The multiple traces as generated above can be combined into a unified trace for profiling using `merge_traces.sh` utility script. The full path for the script is `/opt/rocm/bin/merge_traces.sh`

   Usage:

   ```
   merge_traces.sh -o <outputdir> [<inputdir>...]
   ```

   For example:

   ```
   $ ./merge_traces.sh -h
   ```

   Use the following input arguments to the *merge_traces.sh* script to control which traces are merged and where the resulting merged trace is saved.

   - `<inputdir>...` - space-separated list of ROCProfiler directories to merge. If not specified, the current working directory is used.
   - `-o <outputdir>` - output directory where the results are aggregated. The file `unified/results.json` is generated, and it contains trace data from the specified input directories.

CHAPTER

FOUR

ROCPROF COMMAND HELP

Obtain command line help by typing the following:

```
rocprof -h
```

This returns the following information:

```
RPL: on '240505_115025' from '/opt/rocm-6.2.0-13748' in '/home/rocm'
ROCm Profiling Library (RPL) run script, a part of ROCprofiler library package.
Full path: /opt/rocm-6.2.0-13748/bin/rocprof
Metrics definition: /opt/rocm-6.2.0-13748/lib/rocprofiler/metrics.xml

Usage:
rocprof [-h] [--list-basic] [--list-derived] [-i <input .txt/.xml file>] [-o <output CSV␣
↪file>] <app command line>

Options:
-h - this help
--tool-version <1|2> - to use specific version of rocprof tool, by default v1 is used
            1 - rocprofiler tool v1
            2 - rocprofiler tool v2
--verbose - verbose mode, dumping all base counters used in the input metrics
--list-basic - to print the list of basic HW counters
--list-derived - to print the list of derived metrics with formulas
--cmd-qts <on|off> - quoting profiled cmd line [on]

-i <.txt|.xml file> - input file
    Input file .txt format, automatically rerun application for every profiling features␣
↪line:

        # Perf counters group 1
        pmc : Wavefronts VALUInsts SALUInsts SFetchInsts FlatVMemInsts LDSInsts␣
↪FlatLDSInsts GDSInsts VALUUtilization FetchSize
        # Perf counters group 2
        pmc : WriteSize L2CacheHit
        # Filter by dispatches range, GPU index and kernel names
        # supported range formats: "3:9", "3:", "3"
        range: 1 : 4
        gpu: 0 1 2 3
        kernel: simple Pass1 simpleConvolutionPass2

    Input file .xml format, for single profiling run:
```

(continues on next page)

(continued from previous page)

```
        # Metrics list definition, also the form "<block-name>:<event-id>" can be used
        # All defined metrics can be found in the 'metrics.xml'
        # There are basic metrics for raw HW counters and high-level metrics for derived␣
↪counters
        <metric name=SQ:4,SQ_WAVES,VFetchInsts
        ></metric>

        # Filter by dispatches range, GPU index and kernel names
        <metric
        # range formats: "3:9", "3:", "3"
        range=""
        # list of gpu indexes "0,1,2,3"
        gpu_index=""
        # list of matched sub-strings "Simple1,Conv1,SimpleConvolution"
        kernel=""
        ></metric>

-o <output file> - output CSV file [<input file base>.csv]
-d <data directory> - directory where profiler store profiling data including traces [/
↪tmp]
    The data directory is automatically removed if it is matching the default temporary␣
↪directory.
-t <temporary directory> - to change the temporary directory [/tmp]
    By changing the temporary directory you can prevent removing the profiling data from␣
↪/tmp or enable removing from not '/tmp' directory.
-m <metric file> - file defining custom metrics to use in-place of defaults.

--basenames <on|off> - to turn on/off truncating of the kernel full function names till␣
↪the base ones [off]
--timestamp <on|off> - to turn on/off the kernel dispatches timestamps, dispatch/begin/
↪end/complete during kernel profiling [off]
--ctx-wait <on|off> - to wait for outstanding contexts on profiler exit [on]
--ctx-limit <max number> - maximum number of outstanding contexts [0 - unlimited]
--heartbeat <rate sec> - to print progress heartbeats [0 - disabled]
--obj-tracking <on|off> - to turn on/off kernels code objects tracking [on]
    To support V3 code object

--stats - generating kernel execution stats, file <output name>.stats.csv

--roctx-trace - to enable rocTX application code annotation trace, "Markers and Ranges"␣
↪JSON trace section.
--hip-trace - to trace HIP, generates API execution stats and JSON file chrome-tracing␣
↪compatible
--hsa-trace - to trace HSA, generates API execution stats and JSON file chrome-tracing␣
↪compatible
--sys-trace - to trace HIP/HSA APIs and GPU activity, generates stats and JSON trace␣
↪chrome-tracing compatible
    '--hsa-trace' can be used in addition to select activity tracing from HSA (ROCr␣
↪runtime) level
    Generated files: <output name>.<domain>_stats.txt <output name>.json
    Traced API list can be set by input .txt or .xml files.
```

(continues on next page)

(continued from previous page)

```
    Input .txt:
    hsa: hsa_queue_create hsa_amd_memory_pool_allocate
    Input .xml:
    <trace name="HSA">
        <parameters list="hsa_queue_create, hsa_amd_memory_pool_allocate">
        </parameters>
    </trace>

--roctx-rename - to rename kernels with their enclosing rocTX range's message.

--trace-start <on|off> - to enable tracing on start [on]
--trace-period <dealy:length:rate> - to enable trace with initial delay, with periodic␣
↪sample length and rate
    Supported time formats: <number(m|s|ms|us)>
--flush-rate <rate> - to enable trace flush rate (time period)
    Supported time formats: <number(m|s|ms|us)>
--parallel-kernels - to enable concurrent kernels

Configuration file:
You can set your parameters defaults preferences in the configuration file 'rpl_rc.xml'.␣
↪The search path sequence: .:/home/rocm:<installation directory>
First the configuration file is searched in the current directory, then in the current␣
↪user's home directory, and then in the installation directory.
Configurable options: 'basenames', 'timestamp', 'ctx-limit', 'heartbeat', 'obj-tracking'.
An example of 'rpl_rc.xml':
    <defaults
    basenames=off
    timestamp=off
    ctx-limit=0
    heartbeat=0
    obj-tracking=off
    ></defaults>

--merge-traces - Script for aggregating results from multiple rocprofiler out directries.
                 Usage: if running with rocprof
                 rocprof --merge-traces -o <outputdir> [<inputdir>...]
```

CHAPTER

FIVE

USING ROCPROFV2

Note: `rocprofv2` is considered beta software.

`rocprofv2` is a command-line interface tool (CLI) that lets you profile AMD GPU applications without any requirement of source code modification. The usage of `rocprofv2` along with various command-line arguments is described in the following sections.

To see all the `rocprofv2` options, refer to *rocprofv2 command help*, or run the following from the command line:

```
rocprofv2 --help
```

5.1 Application tracing

Tracing of application and hardware events, is a primary feature of the `rocprofv2` command. The various options for tracing HIP/HSA API, asynchronous activity, and kernel dispatches are described in the following table:

Tracing mode	Option	Usage
HIP API tracing	`--hip-api`	`rocprofv2 --hip-api <app_relative_path>`
Combined HIP API and asynchronous activity tracing	`--hip-activity` or `--hip-trace`	`rocprofv2 --hip-activity <app_relative_path>`
HSA API tracing	`--hip-api`	`rocprofv2 --hsa-api <app_relative_path>`
Combined HSA API and asynchronous activity tracing	`--hip-activity` or `--hsa-trace`	`rocprofv2 --hsa-api <app_relative_path>`
ROCTx API tracing	`--roctx-trace`	`rocprofv2 --roctx-trace <app_relative_path>`
Kernel dispatches tracing	`--kernel-trace`	`rocprofv2 --kernel-trace <app_relative_path>`
All tracing modes combined	`--sys-trace`	`rocprofv2 --sys-trace <app_relative_path>`

Note: By default, the output of these options is directed to `stdout` unless the `-o` option is also specified.

To generate output from these trace options, use one of the supported plugins that generate output in a specific format, as explained in *Formatting output using plugins*. The default plugin is the `file` plugin that generates a CSV file returned to `stdout`, or returned to a file when used with `-o` option.

`rocprofv2` supports API tracing at both HIP and HSA level. In general, HIP APIs directly interact with the user program. It is easier to analyze HIP traces as you can directly map the traces to the program. HSA API tracing is more suited for advanced users who want to understand the application behavior at the lower level.

Both HIP and HSA APIs support asynchronous behavior (e.g., asynchronous memory copy). If trace collection is triggered using either `--hip-api` or `--hsa-api`, the trace records only the start, stop, and duration of API events, but not the execution time of associated actions like memory copy. To record the duration of asynchronous activities, use `--hip-activity` and `--hsa-activity` options, which record both the API events and asynchronous events.

5.1.1 Visualize tracing results

You can view the traces generated by `rocprofv2` using the Perfetto UI that enables you to view and analyze traces in a web browser. To begin go to Perfetto UI, select *Open trace file* from the left-side menu, and select the ROCProfiler trace file to view.

The following is a screenshot from the Perfetto interface. The tasks are organized in a Gantt chart style with the x-axis representing time and each rectangle representing the start and the end time of a task. The tasks are organized in rows. In the figure is the HIP API, HSA API, a queue, and a stream.

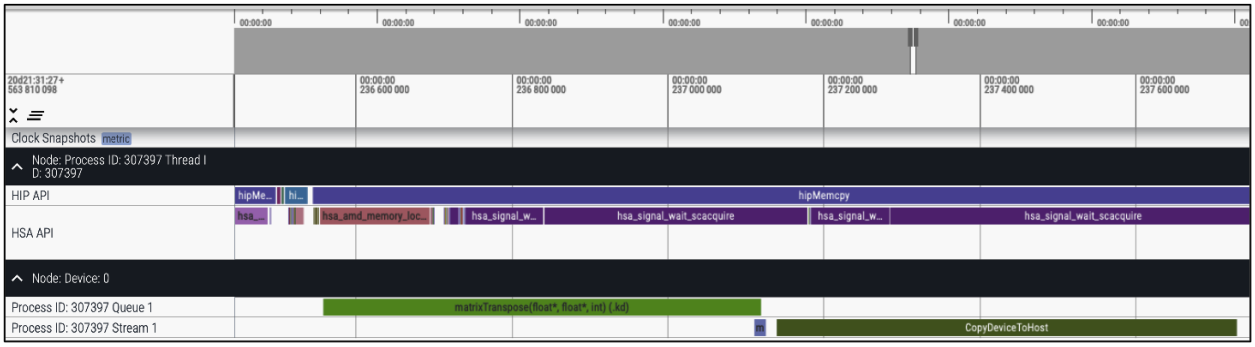

Fig. 5.1: Visualizing Traces Generated Using sys-trace

Tip: To enlarge the image, right click on the image and use the *Open image in new tab* option.

5.2 Kernel profiling

As explained in rocprof-counters application tracing lets you evaluate the timeline of application events, but is little help in providing insight into kernel execution details. The kernel profiling functionality lets you 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 hardware performance.

To check the supported performance counters and metrics, use:

```
rocprofv2 --list-counters
```

The following is a sample output from the `--list-counters` option. The output has been truncated for explanation:

```
gfx1030:0 : SQ_WAVES
: Count number of waves sent to SQs. {emulated, global, C1}
block SQ can only handle 8 counters at a time
```

The fields in the output are:

- **gfx1030:0** - The GPU architecture and GPU ID (separated by colon). The GPU ID needs to be specified as there might be multiple GPUs in the system.
- **SQ_WAVES** - The counter name. Typically, the first token before the first underscore is the GPU block name. Here, SQ is the block that is responsible for managing wavefronts and issuing instructions.

Note: For more information on the performance counters available on AMD GPUs, refer to the GPU architecture documentation.

5.2.1 Input file

To collect basic counters and derived metrics, define the profiling scope in an input file, and specify the file on the command line:

```
rocprofv2 -i input.txt <app_relative_path>
```

An input file is a text file that can be supplied to `rocprofv2` for basic counter and derived metric collection. It contains the list of basic counters or derived metrics to be collected.

Sample Input File:

```
pmc: SQ_WAVES TA_UTIL
```

The fields in the input file are detailed in *Input File*.

PMC: The rows in the text file beginning with `pmc:` are the group of basic counters or derived metrics the user is interested in collecting. The basic counters or derived metrics can be selected from the output generated by `--list-counters` option.

The number of basic counters or derived metrics that can be collected in one run of profiling is limited by the GPU hardware resources. If too many counters/metrics are selected, the kernels need to be executed multiple times to collect the counters/metrics. For multi-pass execution, include multiple rows of `pmc:` in the input file. Counters or metrics in each `pmc:` row can be collected in each run of the kernel.

GPU: The row beginning with the keyword `gpu:` specifies the GPU(s) on which the hardware counters are to be collected. This enables the support for profiling multiple GPUs. You can specify multiple GPUs separated by comma such as `gpu: 1,3`.

Kernel: The row beginning with the `kernel:` keyword specifies the names of kernels to be profiled.

Range: The row beginning with the keyword `range:` specifies the range of kernel dispatches. Specifying range is helpful in cases where the application causes multiple kernel dispatches and users want to filter some kernel dispatches. In the above example, the `range: 0:1` depicts that one kernel is profiled.

5.2.2 Kernel profiling output

This section discusses the kernel profiling output generated using the *Input File*. `rocprofv2` reports one value per metric per kernel in the output. You can generate the output in desired format as described in *Formatting output using plugins*. If no plugin is specified while generating the output, the result is dumped on the command-line.

The following sample output is generated using the `file` plugin. Each row of the file is an instance of kernel execution.

For each kernel, basic information (e.g., `GPU_ID`, `SGPR`, `PID`, etc.) and performance counters (specified in the input file) values are listed. The information is generated in the format of field name and value.

```
$ rocprofv2 -i input.txt --plugin file -o result MatrixTranspose

$ cat results_result.csv

Dispatch_ID,GPU_ID,Queue_ID,Queue_Index,PID,TID,GRD,WGR,LDS,SCR,Arch_VGPR,ACCUM_VGPR,
SGPR,Wave_Size,SIG,OBJ,Kernel_Name,Start_Timestamp,End_Timestamp,Correlation_ID,
SQ_WAVES,GRBM_COUNT,GRBM_GUI_ACTIVE,SQ_INSTS_VALU,FETCH_SIZE

1,64700,1,0,353,353,1048576,16,0,0,8,0,16,64,140356026185088,1,"matrixTranspose(float*,␣
↪float*, int)
(.kd)",7,30064771072,0,65536.000000,398333.000000,398333.000000,917504.000000,4136.000000

2,64700,1,2,353,353,1048576,16,0,0,8,0,16,64,140356026184832,2,"matrixTranspose(float*,
float*, int)
(.kd)",7,30064771072,0,65536.000000,586424.000000,586424.000000,917504.000000,4130.437500

3,64700,1,4,353,353,1048576,16,0,0,8,0,16,64,140356026184576,3,"matrixTranspose(float*,
float*, int)
(.kd)",7,30064771072,0,65536.000000,392460.000000,392460.000000,917504.000000,4129.937500
```

The fields in the output file are:

Output fields	Description
`Dispatch_ID`	Kernel's dispatch Id
`GPU_ID`	GPU identifier to which the kernel was submitted
`Queue_ID`	*ROCm* queue unique identifier to which the kernel was submitted
`Queue_Index`	*ROCm* queue write index for the submitted AQL packet
`PID`	System application process id that submitted the kernel
`TID`	System application thread id that submitted the kernel
`GRD`	Kernel's grid size
`WGR`	Kernel's work group size
`LDS`	Kernel's Local Data Share (LDS) memory size
`SCR`	Kernel's scratch memory size
`Arch_VGPR`	Number of Vector General Purpose Registers (VGPR) used in kernel dispatch
`ACCUM_VGPR`	Total Count of VGPRs
`SGPR`	Kernel's Scalar General-Purpose Register (SGPR) size
`Wave_Size`	Number of wavefronts
`SIG`	Kernel's completion signal
`OBJ`	Code object
`Kernel_Name`	Name of the dispatched kernel
`Start_Timestamp`	Begin time in nanoseconds (ns) when the kernel begins execution
`End_Timestamp`	End time in ns when the kernel finishes execution
`Correlation_ID`	Unique identifier for correlation between HIP and HSA async calls during activity tracing

You can view the generated output using the *Perfetto UI* as previously described in *Visualize Tracing Results*. The following is a screenshot of the Perfetto UI when viewing the kernel profiling output.

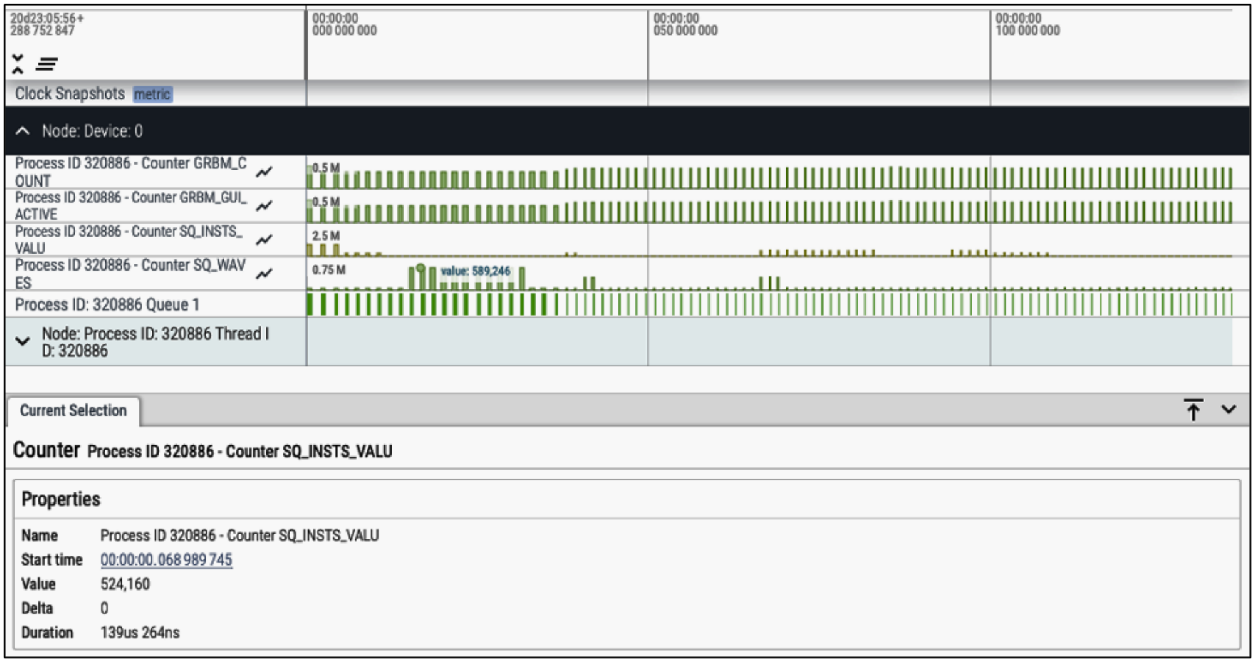

Fig. 5.2: Viewing kernel profiling output

The first four rows represent the performance counters as specified in the input file. The last row is the kernel execution timeline, which is the same as the `--kernel-trace` option used in the *Application tracing* mode.

Viewing the profile results provides a good overview of kernel execution times and how performance metrics values change across the kernels. Additionally, you can also see the exact value of a counter/metric by hovering over or clicking the bar.

5.3 Formatting output using plugins

`rocprofv2` uses a modular plugin system which allows you to generate profiling output in the desired format. Because these plugins are modular in nature, they can easily be decoupled from the code based on need. By default, `rocprofv2` generates the profiling output using the `file` and CLI plugins.

You can install other plugins (as listed in the table below) using the plugins package as shown:

```
rocprofiler-plugins_2.0.0-local_amd64.deb
-or-
rocprofiler-plugins-2.0.0-local.x86_64.rpm
```

You can also create your own plugins if you are using `rocprofv2` with source code and not just as a CLI tool. To write new plugins import the `include/rocprofiler/v2/rocprofiler_plugins.h` header file.

To generate the profiling output using a plugin, use:

```
rocprofv2 --plugin plugin_name -i input.txt <app_relative_path>

# where plugin_name is file, perfetto, att, or ctf
```

To specify the plugin version to be used in case of multiple versions, use:

```
rocprofv2 --plugin <plugin_name> --plugin-version <plugin_version_required> <rocprofv2_
↪options> <app_relative_path>
```

The following table lists the available plugins:

Plugin	Output format
File	Text files (.csv or .txt)
Perfetto	Protobuf in the format of the Chromium Project's trace-event format
Advanced Thread Tracer (ATT)	Binary and .csv formats for *Analysis View tool*
Common Trace Format (CTF)	Binary, formatted in the ctf format that can be consumed by public tools such as Babeltrace and TraceCompass

Note: To generate output, the plugins require you to set the *OUTPUT_PATH* variable to the desired directory. File plugin is the only plugin that still generates output in the absence of *OUTPUT_PATH* by dumping the output to standard output.

5.3.1 File plugin

To output the data in `.txt` files using file plugin, use:

```
rocprofv2 --plugin file -i samples/input.txt -d output_dir <app_relative_path>
```

Note that specifying the directory for output files using `-d` is optional.

File plugin has two versions with version 2 being the default. The headers in the output files generated using file plugin version 1 and 2 differ as shown below.

Version 1 header:

```
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,arch_vgpr,accum_
↪vgpr,sgpr,wave_size,sig,obj,DispatchNs,BeginNs,EndNs,CompleteNs,Counters
```

Note that the version 1 header is same as the legacy `rocprof` output.

Version 2 header:

```
Dispatch_ID,GPU_ID,Queue_ID,PID,TID,Grid_Size,Workgroup_Size,LDS_Per_Workgroup,Scratch_
↪Per_Workitem,Arch_VGPR,Accum_VGPR,SGPR,Wave_Size,Kernel_Name,Start_Timestamp,End_
↪Timestamp,Correlation_ID,Counters
```

5.3.2 Perfetto plugin

To output the data in *Protobuf* format using the Perfetto plugin, use:

```
rocprofv2 --plugin perfetto --hsa-trace <app_relative_path>
```

You can view the *Protobuf* files using Perfetto or Trace processor.

5.3.3 Common Trace Format plugin

To output the data in Common Trace Format (CTF), which is a binary trace format, use:

```
rocprofv2 --plugin ctf --hip-trace <app_relative_path>
```

You can view the CTF binary output using TraceCompass or Babeltrace.

For information on the ATT plugin, refer to the *Analysis view*.

5.4 Analysis View

Analysis View is a web-based tool that allows you to visualize detailed information within a running kernel. It allows you to dive deeper into the kernels and examine how metrics change over time. This is more helpful for analyzing kernel execution than the standard kernel profiling which reports only one value for a counter for a kernel.

Analysis View supports collection of the following data:

- Fast counter (supported only in *GFX9* product family)
- Wave states
- Instruction delays
- Memory barrier dependencies

For each run, data can be collected from one Compute Unit (CU). A user can choose any CU from 0 to 15 for data collection in Analysis View. The selected CU is 1 by default.

5.4.1 Running the Analysis View

This section provides the information required to run the Analysis View tool.

Prerequisites

Install the Python packages `numpy`, `matplotlib`, and `websockets`.

```
python3 -m pip install numpy matplotlib websockets
```

Compiling the application

Analysis View requires assembly files for the kernel being profiled. You can generate these files using either of these two options:

- Individually for the application

```
hipcc -g --savetemps
```

- Set environment variable `HIPCC_COMPILE_FLAGS_APPEND` to configure the assembly files to be generated for all applications compiled by `hipcc`.

```
export HIPCC_COMPILE_FLAGS_APPEND="--save-temps -g"
```

Supplying input file

To specify options for information collection using Analysis View, configure the input file using the parameters given below. You can specify the values in the input file as decimal (for example, 15) or hex (for example, 0xF).

Table 5.1: input file

Parameter	Need for specification	Description
`att: TARGET_`	Required	Analysis View data collection retrieves information related to instruction timing on a single CU. This parameter must be the first string in the configuration file to mark the CU from which the data needs to be retrieved. Allowed values are in the range (0, 15). You can select any CU from 1 to 15, however we recommend selecting CU 1 for better results.
`SIMD_MA`	Optional	This is a binary mask representing the SIMDs that generate instruction data on the given `TARGET_CU`. Allowed values are in the range (0x0, 0xF). Recommended value is 0x3. Note that Analysis View generates a lot of data and if you encounter packet loss, it is better to reduce the value of `SIMD_MASK`.
`KERNEL`	Optional	This parameter adds a kernel filter. Only kernels containing the given string are profiled in Analysis View. Used for large applications, where profiling every kernel generates a lot of Analysis View data. Adding multiple filters is supported.
`PERFCOU`	Required only for performance counter collection	This parameter defines the frequency at which counters need to be collected. Allowed values are in the range (0, 31), with 0 being the fastest and 31 being the slowest. Recommended value is 3.
`PERFCOU`	Optional	This parameter enables basic counter collection for the given name. Only SQ block counters can be used. For example, `SQ_BUSY_CU_CYCLES` and maximum 16 counters can be specified.
`PERFCOU`	Optional	This parameter performs the same function as `PERFCOUNTER` with the counter ID being specified instead of the counter name.

Example: A simple *input.txt* file sample

```
att: TARGET_CU=1
SIMD_MASK=0x1
```

Example: *input.txt* file sample with kernel filters

```
att: TARGET_CU=1
SIMD_MASK=0x3
KERNEL=vectoradd
KERNEL=histogram
```

Specifying the kernel in the above input file ensures that only kernels with *vectoradd* or *histogram* in their name are collected.

Example: *input.txt* file sample for performance counter collection

```
att: TARGET_CU=1
SIMD_MASK=0x3
PERFCOUNTERS_COL_PERIOD=0x2
PERFCOUNTER=SQ_BUSY_CU_CYCLES
PERFCOUNTER=SQ_WAVE_CYCLES
PERFCOUNTER=SQ_WAIT_ANY
```

Starting Analysis View

To start Analysis View, run `rocprofv2` command-line interface along with ATT plugin and arguments as shown:

```
rocprofv2 <rocprofv2_args> --plugin att <assembly_file> <att_args> <application>
```

To learn about the values that can be passed to the command-line arguments given in the command above, see the table below:

Table 5.2: analysis view arguments

Com line ar- gu- ment	Values	Re- quire	Description
<rocp	`-i <input_file> -d <output_locatic`	Yes Op- tiona	Specifies the input file Specifies the location of the output folder for Analysis View data collection. Default value is "./".
<as- sem- bly_f	Path for the assembly file generated by `hipcc`, which contains the relevant kernel	Yes	The assembly files are generally in the form **amdgcn*.s*. For example, `vectoradd_hip-hip-amdgcn-amd-amdhsa-gfx908.s` for the `vectoradd_hip.cpp` sample on a MI100 GPU.
<att_	`--ports --target --att_ke --trace_ --genasm`	All `att` ar- gu- ment are op- tiona	Specifies a pair of ports to open the viewer on the browser. Default value is 8000,18000 Overrides `TARGET_CU` in the input file. Default value is None. Specifies the kernel to be loaded on the viewer. Default value is *output_path/*kernel.txt*. Loads all the data present in the output folder. Specifies the trace file (.att) to be used. Uses the files from `att_kernel` by default. Specifies the filename (**.s*) to be generated with looped `waitcnt` barriers. This option is rather unique. Typically, `waitcnt` instruction waits for a group of instructions in one call and the number of instructions a `waitcnt` instruction waits for is supplied as its operand. Specifying `--genasm` generates `waitcnt` for one instruction per `waitcnt`. This allows you to view individual instruction delays. Default value is None.
<ap- pli- ca- tion>	Name of the application to be profiled	Op- tiona	Specify this option for collecting new Analysis View data. For viewing previously collected data, omit this option.

Running sample application

Follow the steps given below to run an application for Analysis View collection. Note that the use of *vectorAdd* HIP application is for demonstration purpose only.

1. Go to the directory that contains the application.

```
cd HIP-Examples-master/vectorAdd/
```

2. Create *input.txt* file with the following content to specify a target CU and SIMD mask.

```
att: TARGET_CU=1
SIMD_MASK=0x3
```

3. Compile the HIP application while generating the assembly files (**.s*).

```
hipcc -g --save-temps vectoradd_hip.cpp -o vectoradd_hip.exe
```

4. Execute `rocprofv2` command by specifying the *input.txt* and ISA file.

```
/opt/rocm/bin/rocprofv2 -i input.txt --plugin att *amdgcn*.s  ./vectoradd_hip.exe
```

Generated Output

On successful Analysis View data collection and analysis, the following message is displayed:

"Serving at ports: 8000,18000"

These files are generated:

- ***kernel.txt**: It contains the mangled and demangled name of the kernel that was profiled.
- ***.att**: It contains the data per shader engine.

Each profiled kernel generates its own **.att* and *kernel.txt* files. If data from multiple kernels with the same name is generated from different runs or due to the kernel running in a loop, the file name gets incremented (v0, v1, v2, and so on) and the viewer allows you to choose the run to be displayed.

You can view the collected data in the browser at *localhost:8000*. Make sure to use port forwarding when running on a remote machine.

5.4.2 Types of views

Analysis View supports three views, including the *Hotspot view*, Instruction view, and *SIMD view*. It also generates a *graph for performance counter collection*. The views are described in detail below.

Hotspot view

The hotspot view is used to quickly find the instructions with most cycles used. It allows you to see a histogram that shows cycles used by grouped sections of code. Clicking on a line puts the given instruction into view.

Instruction view

The instruction view shows the number of cycles taken by each instruction to complete. It is used to link trace data to the underlying assembly code. Hovering the mouse over an instruction displays the number of cycles taken by the instruction to be issued and complete. If an instruction runs multiple times (in a loop), the cycles are shown for the first run. Clicking on an instruction puts the corresponding instruction into view in the wave bar.

SIMD view

The Single Instruction Multiple Data (SIMD) view shows the timeline for waves belonging to a given SIMD and CU. Each wave slot contains two bars, the instruction and timeline bar. Each rectangle is an instruction color coded as per its type, which allows you to quickly see the general SIMD/CU usage. Hovering over the rectangle displays the instruction type and the current clock cycle. Along with the instruction type, each wave has a timeline bar, where red indicates *stall* (stalled), green indicates *exec* (ready to execute), yellow indicates *wait* (waiting usually for barriers), and white indicates *empty* (no wave allocated).

Counters graph

A graph is generated for performance counters if they are collected. Performance counters are helpful in providing GPU-wide information. This information is averaged over all shader engines and over the number of quad cycles (polling rate). Note that some counters display values multiplied by 4, such as BUSY CUs, as they increment differently than other counters. Also, normalization causes the counter values to be divided by their maximum value and be shown as percentage of maximum (checked by default). You have the option to select the counters to be visualized and normalize the graph.

If no counters are present, the graph shows waves states instead. Wave states are low pass filtered for improved visualization.

Running on a remote machine

Analysis View allows you to profile an application running on a remote server over `ssh`. You can achieve this in two simple steps:

1. Generate the trace data and assembly files on the remote server.
2. Download all the data (**kernel.txt*, **.att*, **.s*) on the local machine. You can view all the traces locally using a locally installed `rocprofv2`.

Alternatively, since the Analysis View is a web-based tool, you can use `ssh` port forwarding:

```
ssh -L 8000:localhost:8000 <user@IP>
```

CHAPTER

SIX

ROCPROFV2 COMMAND HELP

Obtain command line help by typing the following:

```
rocprofv2 -h
```

This returns the following information:

```
ROCProfilerV2 Run Script Usage:
-h   | --help                 For showing this message
--list-counters               For showing all available counters for the current GPUs
-m                            For providing an absolute path of a custom metrics file
--basenames                   For Truncating the kernel names
--hip-api                     For Collecting HIP API Traces
--hip-activity | --hip-trace  For Collecting HIP API Activities Traces
--hsa-api                     For Collecting HSA API Traces
--hsa-activity | --hsa-trace  For Collecting HSA API Activities Traces
--roctx-trace                 For Collecting ROCTx Traces
--kernel-trace                For Collecting Kernel dispatch Traces
--sys-trace                   For Collecting HIP and HSA APIs and their Activities␣
↪Traces along ROCTX and Kernel Dispatch traces

    #usage e.g: rocprofv2 --[hip-trace|hsa-trace|roctx-trace|kernel-trace|sys-trace]
↪<executable>

--plugin  PLUGIN_NAME         For enabling a plugin (cli/file/perfetto/att/ctf)
    # usage(file/perfetto/ctf) e.g: rocprofv2 -i pmc.txt --plugin [file/perfetto/ctf] -d␣
↪out_dir <executable>
    # usage(att): rocprofv2 <rocprofv2_params> --plugin att <ISA_file> <att_parameters>
↪<executable>
    # use "rocprofv2 --plugin att --help" for ATT-specific parameters help.

--plugin-version  <1|2>       For selecting the version for the plugin (1/2)
    # 1 - Legacy output format, 2 - New output format (default)

-i   | --input                For adding counters file path (every line in the text file␣
↪represents a counter)
    # usage: rocprofv2 -i pmc.txt -d <executable>

-o   | --output-file-name          For the output file name
    # usage e.g:(with current dir): rocprofv2 --hip-trace -o <file_name> <executable>
    # usage e.g:(with custom dir):  rocprofv2 --hip-trace -d <out_dir> -o <file_name>
↪<executable>
```

(continues on next page)

(continued from previous page)

```
-d   | --output-directory     For adding output path where the output files will be saved
    # usage e.g:(with custom dir):  rocprofv2 --hip-trace -d <out_dir> <executable>

-fi  | --flush-interval       For adding a flush interval in milliseconds, every "flush␣
↪interval" the buffers will be flushed
    # usage e.g:  rocprofv2 --hip-trace -fi 1000 <executable>

-tp  | --trace-period        Specifies a trace period in milliseconds, with format "-tp
↪<DELAY>:<ACTIVE_TIME>:<LOOP_RESET_TIME>".
    # usage e.g:  rocprofv2 --hip-trace -tp 1000:2000:4000 <executable>
```

CHAPTER

SEVEN

USING ROCSYS

`rocsys` is a command-line utility tool used to invoke and control a profiling session (launch/start/stop/exit) on an application being traced or profiled. `rocsys` is especially useful for selective profiling of applications with long-running workloads (such as DNN training) as it allows you to profile and control the application while it is running. You can also launch the session from one terminal and control the application using `rocsys` from another terminal.

To see all the `rocsys` options, run:

```
rocsys -help

rocsys: launch must be preceded by --session <name>
e.g. rocsys --session <SESSION_NAME> launch <MPI_COMMAND> <MPI_ARGUMENTS> rocprofv2
<ROCPROFV2_OPTIONS> <APP_EXEC>

where all mpiexec options must come before rocsys
rocsys: start must be preceded by --session <name>
   rocsys --session <name> start
rocsys: stop must be preceded by --session <name>
   rocsys --session <name> stop
rocsys: exit must be preceded by --session <name>
   rocsys --session <name> exit
```

The following are the session management options used with `rocsys` in the given order to achieve selective profiling on the `rocprofv2` run:

1. **Launch** - Creates a session. After launching the application stops until the session is started as shown in step 2.

```
/opt/rocm/bin/rocsys --session session1 launch rocprofv2 -i ../samples/input.txt <long_
↪running_app>
ROCSYS:: Session ID: 2109
ROCSYS Session Created!
ROCProfilerV2: Collecting the following counters:
- SQ_WAVES
- GRBM_COUNT
- GRBM_GUI_ACTIVE
- SQ_INSTS_VALU
- FETCH_SIZE
Enabling Counter Collection
```

2. **Start** - Starts the halted after launching session on the same or another terminal, and begins dumping kernel profiling information. The start command triggers the halted application to run.

```
/opt/rocm/bin/rocsys --session session1 start
ROCSYS:: Starting Tools Session...
Dispatch_ID(1), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(2), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(3), GPU_ID(1), ... // All the metrics of a kernel
```

3. **Stop** - Stops the session. The information displayed on the terminal is a result of kernel profiling between the current and the previous `rocsys` command. Note that this command stops only the profiling session without affecting the application on the run.

```
/opt/rocm/bin/rocsys --session session1 stop
ROCSYS:: Stopping Tools Session...
Dispatch_ID(22397), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(22398), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(22399), GPU_ID(1), ... // All the metrics of a kernel
```

4. **Start** (to restart) - rocsys allows you to start and stop the session innumerable times once the session is created. This helps in analyzing batches of kernel profiling information.

```
/opt/rocm/bin/rocsys --session session1 start
ROCSYS:: Starting Tools Session...
Dispatch_ID(22400), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(22401), GPU_ID(1), ... // All the metrics of a kernel
```

5. **Exit** - Exits the profiling session. Once the session is exited, it cannot be restarted.

```
/opt/rocm/bin/rocsys --session session1 exit
Dispatch_ID(16828), GPU_ID(1), ... // All the metrics of a kernel
Dispatch_ID(16829), GPU_ID(1), ... // All the metrics of a kernel
ROCSYS:: Exiting Tools Session...Application might still be finishing up..
```

Note: Exiting the session only stops profiling. The application could continue running to completion in the background. If you don't want to wait for the application to finish, use `CTRL+C` to stop the application after `exit`.

CHAPTER

EIGHT

ROCPROFILER LIBRARY SPECIFICATION

The goal of the implementation is to provide a hardware-specific low-level performance analysis interface for profiling of GPU compute applications. The profiling includes hardware performance counters with complex performance metrics and HW traces. Performance counters are treated as the basic metrics, and formulas can be defined to derive more complex metrics.

The library can be loaded by HSA runtime as a tool plugin, and it can be loaded by higher level hardware-independent performance analysis API like PAPI. The library provides a C API and is based on AQLprofile AMD specific HSA extensions.

1. The library provides methods to query the list of supported HW features.
2. The library provides profiling APIs to start, stop, read metrics results and tracing data.
3. The library provides intercepting API for collecting per-kernel profiling data for the kernels dispatched to HSA AQL queues.
4. The library provides mechanism to load profiling tool library plugin by env variable ROCP_TOOL_LIB.
5. The library is responsible for allocation of the buffers for profiling and notifying about output data buffer overflow for traces.
6. The library is implemented based on AMD specific AQLprofile HSA extension.
7. The library implementation is abstracted from the specific GFXIP.
8. The library implementation is extensible:
    - Easy adding of counters and metrics
    - Counters enumeration
    - Counters and metrics can be dynamically configured using XML configuration files (`metrics.xml`) with counters and metrics tables:
        - Counters table entry, basic metric: counter name, block name, event id
        - Complex metrics table entry: metric name, an expression for calculation the metric from the counters

Metrics XML file example:

```
<gfx8>
    <metric name=L1_CYCLES_COUNTER block=L1 event=0 >
    <metric name=L1_MISS_COUNTER block=L1 event=33 >
    ...
</gfx8>

<gfx9>
    ...
```

(continues on next page)

(continued from previous page)

```
</gfx9>

<global>
    <metric name=L1_MISS_RATIO expr=L1_CYCLES_COUNT/ L1_MISS_COUNTER ></metric>
</global>
```

8.1 Environment variables

- `HSA_TOOLS_LIB` - required to be set to the name of rocprofiler library to be loaded by HSA runtime
- `ROCP_METRICS` - path to the metrics XML file
- `ROCP_TOOL_LIB` - path to profiling tool library loaded by ROCProfiler
- `ROCP_HSA_INTERCEPT` - if set then HSA dispatches intercepting is enabled

8.2 Logging

Set the following environment variables to enable logging:

Environment Variable	Purpose	Log Files
`ROCPROFILER_LOG=1`	Enables error message logging	`/tmp/rocprofiler_log.txt`
`ROCPROFILER_TRACE=1`	Enables verbose tracing	`/tmp/roctracer_log.txt`

8.3 General API

The library supports a method for getting the error number and error string of the last failed library API call. To check the conformance of the library API header, and the library binary, the following version macros and API methods can be used.

Returning the error and error string methods:

- `rocprofiler_error_string` - method for returning the error string

Library version:

- `ROCPROFILER_VERSION_MAJOR` - API major version macro
- `ROCPROFILER_VERSION_MINOR` - API minor version macro
- `rocprofiler_version_major` - library major version
- `rocprofiler_version_minor` - library minor version

8.3.1 Returning the error and error string methods

```
const char* rocprofiler_error_string();
```

8.3.2 Library version

The library provides back compatibility if the library major version is less or equal then the API major version macro.

API version macros defined in the library API header `rocprofiler.h`:

```
ROCPROFILER_VERSION_MAJOR
ROCPROFILER_VERSION_MINOR
```

Methods to check library major and minor versions:

```
uint32_t rocprofiler_major_version();
uint32_t rocprofiler_minor_version();
```

8.4 Backend API

The library provides methods to open/close a profiling context, to start, stop, and read HW performance counters and traces, to intercept kernel dispatches to collect per-kernel profiling data. The library also provides methods to calculate complex performance metrics and to query the list of available metrics.

The library distinguishes two profiling features, metrics and traces, where HW performance counters are treated as the basic metrics. To check if there was an error the library methods return HSA standard status code. For a given context the profiling can be started/stopped and counters sampled in standalone mode or profiling can be initiated by intercepting the kernel dispatches with registering a dispatch callback.

For counters sampling, which is the usage model of higher level APIs like PAPI, the start/stop/read APIs should be used. To collect per-kernel data for kernels submitted to HSA queues, the dispatch callback API should be used.

The library provides back compatibility if the library major version is less or equal.

Returned API status:

- `hsa_status_t` - HSA status codes are used from hsa.h header

Loading and Configuring, loadable plugin on-load/unload methods:

- `rocprofiler_settings_t` - global properties
- `OnLoadTool`
- `OnLoadToolProp`
- `OnUnloadTool`

Info API:

- `rocprofiler_info_kind_t` - profiling info kind
- `rocprofiler_info_query_t` - profiling info query
- `rocprofiler_info_data_t` - profiling info data
- `rocprofiler_get_info` - return the info for a given info kind
- `rocprofiler_iterote_inf_` - iterate over the info for a given info kind

- rocprofiler_query_info - iterate over the info for a given info query

Context API:

- rocprofiler_t - profiling context handle
- rocprofiler_feature_kind_t - profiling feature kind
- rocprofiler_feature_parameter_t - profiling feature parameter
- rocprofiler_data_kind_t - profiling data kind
- rocprofiler_data_t - profiling data
- rocprofiler_feature_t - profiling feature
- rocprofiler_mode_t - profiling modes
- rocprofiler_properties_t - profiler properties
- rocprofiler_open - open new profiling context
- rocprofiler_close - close profiling context and release all allocated resources
- rocprofiler_group_count - return profiling groups count
- rocprofiler_get_group - return profiling group for a given index
- rocprofiler_get_metrics - method for calculating the metrics data
- rocprofiler_iterate_trace_data - method for iterating output trace data instances
- rocprofiler_time_id_t - supported time value ID enumeration
- rocprofiler_get_time - return time for a given time ID and profiling timestamp value

Sampling API:

- rocprofiler_start - start profiling
- rocprofiler_stop - stop profiling
- rocprofiler_read - read profiling data to the profiling features objects
- rocprofiler_get_data - wait for profiling data

  Group versions of start/stop/read/get_data methods:

  - rocprofiler_group_start
  - rocprofiler_group_stop
  - rocprofiler_group_read
  - rocprofiler_group_get_data

Intercepting API:

- rocprofiler_callback_t - profiling callback type
- rocprofiler_callback_data_t - profiling callback data type
- rocprofiler_dispatch_record_t - dispatch record
- rocprofiler_queue_callbacks_t - queue callbacks, dispatch/destroy
- rocprofiler_set_queue_callbacks - set queue kernel dispatch and queue destroy callbacks
- rocprofiler_remove_queue_callbacks - remove queue callbacks

Context pool API:

- rocprofiler_pool_t - context pool handle
- rocprofiler_pool_entry_t - context pool entry
- rocprofiler_pool_properties_t - context pool properties
- rocprofiler_pool_handler_t - context pool completion handler
- rocprofiler_pool_open - context pool open
- rocprofiler_pool_close - context pool close
- rocprofiler_pool_fetch - fetch and empty context entry to pool
- rocprofiler_pool_release - release a context entry
- rocprofiler_pool_iterate - iterated fetched context entries
- rocprofiler_pool_flush - flush completed context entries

8.4.1 Loading and configuring

The profiling properties can be set by profiler plugin on loading by ROC runtime. The profiler library plugin can be set by ROCP_TOOL_LIB env var.

Global properties:

```
typedef struct {
    uint32_t intercept_mode;
    uint64_t timeout;
    uint32_t timestamp_on;
} rocprofiler_settings_t;
```

On load/unload methods defined in profiling tool library loaded by ROCP_TOOL_LIB env var:

```
extern "C" void OnLoadTool();
extern "C" void OnLoadToolProp(rocprofiler_settings_t* settings);
extern "C" void OnUnloadTool();
```

8.4.2 Info API

The profiling metrics are defined by name and the traces are defined by name and parameters. All supported features can be iterated using iterate_info/query_info methods. The counter names are defined in counters table configuration file, each counter has a unique name and defined by block name and event ID. The traces and trace parameters names are the same as in the hardware documentation and the parameters codes are rocprofiler_feature_parameter_t values, see below in the *Context API* section.

Profiling info kind:

```
typedef enum {
    ROCPROFILER_INFO_KIND_METRIC = 0,               // metric info
    ROCPROFILER_INFO_KIND_METRIC_COUNT = 1,         // metrics count
    ROCPROFILER_INFO_KIND_TRACE = 2,                // trace info
    ROCPROFILER_INFO_KIND_TRACE_COUNT = 3,          // traces count
} rocprofiler_info_kind_t;
```

Profiling info data:

```
typedef struct {
    rocprofiler_info_kind_t kind;                   // info data kind
    union {
        struct {
            const char* name;               // metric name
            uint32_t instances;             // instances number
            const char* expr;               // metric expression, NULL for basic counters
            const char* description;        // metric description
            const char* block_name;         // block name
            uint32_t block_counters;        // number of block counters
        } metric;
        struct {
            const char* name;               // trace name
            const char* description;        // trace description
            uint32_t parameter_count;       // supported by the trace number
                                            // parameters
        } trace;
    };
} rocprofiler_info_data_t;
```

Return info for a given info kind:

```
has_status_t rocprofiler_get_info(
    const hsa_agent_t* agent,                       // [in] GPU handle, NULL for all
                                                    // GPU agents
    rocprofiler info_kind_t kind,                   // kind of iterated info
    void *data);                                    // data passed to callback
```

Iterate over the info for a given info kind, and invoke an application-defined callback on every iteration:

```
has_status_t rocprofiler_iterate_info(
    const hsa_agent_t* agent,                       // [in] GPU handle, NULL for all
                                                    // GPU agents
    rocprofiler info_kind_t kind,                   // kind of iterated info
    hsa_status_t (*callback)(const rocprofiler_info_data_t info, void *data), // callback
    void *data);
```

Iterate over the info for a given info query, and invoke an application-defined callback on every iteration. The query fields set to NULL define the query wildcard:

```
has_status_t rocprofiler_query_info(
    const hsa_agent_t* agent,                       // [in] GPU handle, NULL for all
                                                    // GPU agents
    rocprofiler info_kind_t kind,                   // kind of iterated info
    rocprofiler_info_data_t query,                  // info query
    hsa_status_t (*callback)(const rocprofiler_info_data_t info, void *data), // callback
    void *data);                                    // data passed to callback
```

8.4.3 Context API

Profiling context is accumulating all profiling information including profiling features which carry profiling data, required buffers for profiling command packets and output data. The context can be created and deleted by the library open/close methods.

By deleting the context all accumulated by the library resources associated with this context will be released. If more than one run is required to collect all requested counters data, then data for all profiling groups should be collected first, then the metrics can be calculated by loading the saved groups' data to the profiling context. Saving and loading of the groups data is responsibility of the tool. The groups are automatically identified on the profiling context open and there is API to access them, as shown in the *Profiling Groups* example that follows.

Profiling context handle:

```
typename rocprofiler_t;

Profiling feature kind:

typedef enum {
    ROCPROFILER_FEATURE_KIND_METRIC = 0,    // metric
    ROCPROFILER_FEATURE_KIND_TRACE = 1      // trace
} rocprofiler_feature_kind_t;
```

Profiling feature parameter:

```
typedef hsa_ven_amd_aqlprofile_parameter_t rocprofiler_feature_parameter_t;
```

Profiling data kind:

```
typedef enum {
    ROCPROFILER_DATA_KIND_UNINIT = 0,       // data uninitialized
    ROCPROFILER_DATA_KIND_INT32 = 1,        // 32bit integer
    ROCPROFILER_DATA_KIND_INT64 = 2,        // 64bit integer
    ROCPROFILER_DATA_KIND_FLOAT = 3,        // float single-precision result
    ROCPROFILER_DATA_KIND_DOUBLE = 4,       // float double-precision result
    ROCPROFILER_DATA_KIND_BYTES = 5         // trace output as a bytes array
} rocprofiler_data_kind_t;
```

Profiling data:

```
typedef struct {
    rocprofiler_data_kind_t kind;           // result kind
    union {
        uint32_t result_int32;              // 32bit integer result
        uint64_t result_int64;              // 64bit integer result
        float result_float;         // float single-precision result
        double result_double;               // float double-precision result
        typedef struct {
            void* ptr;              // pointer
            uint32_t size;          // byte size
            uint32_t instances;     // number of trace instances
        } result_bytes;                     // data by ptr and byte size
    };
} rocprofiler_data_t;
```

Profiling feature:

```
typedef struct {
    rocprofiler_feature_kind_t type;                          // feature type
    const char* name;                                         // feature name
    const rocprofiler_feature_parameter_t* parameters;        // feature parameters
    uint32_t parameter_count;                                 // feature parameter count
    rocprofiler_data_t* data;                                 // profiling data
} rocprofiler_feature_t;
```

Profiling mode masks:

There are several modes which can be specified for the profiling context. `STANDALONE` mode can be used for the counters sampling in another then application context to support statistical system-wide profiling. In this mode the profiling context supports its own queue which can be created on the context open if the `CREATEQUEUE` mode is also specified. See the *Profiler Properties* example that follows for the standalone mode queue properties. The profiler supports several profiling groups for collecting profiling data in several runs and `SINGLEGROUP` mode allows only one group and the context open will fail if more groups are needed.

```
typedef enum {
    ROCPROFILER_MODE_STANDALONE = 1,          // standalone mode when ROC profiler
                                              // supports own AQL queue
    ROCPROFILER_MODE_CREATEQUEUE = 2,         // profiler creates queue in STANDALONE mode
    ROCPROFILER_MODE_SINGLEGROUP = 4          // profiler allows one group only and fails
                                              // if more groups are needed
} rocprofiler_mode_t;
```

Context data readiness callback:

```
typedef void (*rocprofiler_context_callback_t)(
    rocprofiler_group_t* group,               // profiling group
    void* arg);                               // callback arg
```

Profiler properties:

There are several properties which can be specified for the context. A callback can be registered which will be called when the context data is ready. In standalone profiling mode `ROCPROFILER_MODE_STANDALONE` the context supports its own queue, and the queue can be set by the property `queue` or a queue will be created with the specified depth `queue_depth` if mode `ROCPROFILER_MODE_CREATEQUEUE` is also specified.

```
typedef struct {
    rocprofiler_context_callback_t callback; // callback on the context data readiness
    void* callback_arg;                       // callback arg
    has_queue_t* queue;                       // HSA queue for standalone mode
    uint32_t queue_depth;                     // created queue depth, for create-queue␣
↪mode
} rocprofiler_properties_t;
```

Open/close profiling context:

```
hsa_status_t rocprofiler_open(
    hsa_agent_t agent,                        // GPU handle
    rocprofiler_feature_t* features,          // [in/out] profiling feature array
    uint32_t feature_count,                   // profiling feature count
    rocprofiler_t** context,                  // [out] profiling context handle
    uint32_t mode,                            // profiling mode mask
```

(continues on next page)

(continued from previous page)

```
    rocprofiler_properties_t* properties);  // profiler properties

hsa_status_t rocprofiler_close(
    rocprofiler_t* context);                // [in] profiling context
```

Profiling groups:

The profiler on the context open automatically identifies a required number of the application runs to collect all data needed for all specified metrics, and creates a metric group for each run. Data for all profiling groups should be collected, and then the metrics can be calculated by loading the saved groups' data to the profiling context. Saving and loading of the groups data is the responsibility of the tool.

```
typedef struct {
    uint32_t index;                         // profiling group index
    rocprofiler_feature_t** features;       // profiling features array
    uint32_t feature_count;                 // profiling feature count
    rocprofiler_t* context;                 // profiling context handle
} rocprofiler_group_t;
```

Return profiling groups count:

```
hsa_status_t rocprofiler_group_count(
    rocprofiler_t* context);                // [in/out] profiling context
    uint32* count);                         // [out] profiling groups count
```

Return the profiling group for a given index:

```
hsa_status_t rocprofiler_get_group(
    rocprofiler_t* context,                 // [in/out] profiling context,
                                            // will be returned as
                        // a part of the group structure
    uint32_t index,                         // [in] group index
    rocprofiler_group_t* group);            // [out] profiling group
```

Calculate metrics data. The data will be stored to the registered profiling features data fields. After all profiling context data is ready the registered metrics can be calculated. The context data readiness can be checked by `get_data` API or using the context callback.

```
hsa_status_t rocprofiler_get_metrics(
    rocprofiler_t* context);                // [in/out] profiling context
```

Method for iterating trace data instances:

Trace data can have several instances, for example, one instance per Shader Engine.

```
hsa_status_t rocprofiler_iterate_trace_data(
    const rocprofiler_t* contex,                    // [in] context object
    hsa_ven_amd_aqlprofile_data_callback_t callback, // [in] callback to iterate
                                                    // the output data
    void* callback_data);                           // [in/out] passed to callback data
```

Converting of profiling timestamp to time value for supported time ID.

Supported time value ID enumeration:

```
typedef enum {
ROCPROFILER_TIME_ID_CLOCK_REALTIME = 0,   // Linux realtime clock time
ROCPROFILER_TIME_ID_CLOCK_MONOTONIC = 1,  // Linux monotonic clock time
} rocprofiler_time_id_t;
```

Method for converting of profiling timestamp to time value for a given time ID:

```
hsa_status_t rocprofiler_get_time(
rocprofiler_time_id_t time_id,            // identifier of the particular
                                            // time to convert the timestamp
uint64_t timestamp,                       // profiling timestamp
uint64_t* value_ns);                      // [out] returned time 'ns' value
```

8.4.4 Sampling API

The API supports the counters sampling usage model with start/read/stop methods and also lets to wait for the profiling data in the intercepting usage model with get_data method.

Start/stop/read methods:

```
hsa_status_t rocprofiler_start(
    rocprofiler_t* context,               // [in/out] profiling context
    uint32_t group_index = 0);            // group index

hsa_status_t rocprofiler_stop(
    rocprofiler_t* context,               // [in/out] profiling context
    uint32_t group_index = 0);            // group index

hsa_status_t rocprofiler_read(
    rocprofiler_t* context,               // [in/out] profiling context
    uint32_t group_index = 0);            // group index
```

Wait for profiling data:

```
hsa_status_t rocprofiler_get_data(
    rocprofiler_t* context,               // [in/out] profiling context
    uint32_t group_index = 0);            // group index
```

Group versions of the above start/stop/read/get_data methods:

```
hsa_status_t rocprofiler_group_start(
    rocprofiler_group_t* group);          // [in/out] profiling group

hsa_status_t rocprofiler_group_stop(
    rocprofiler_group_t* group);          // [in/out] profiling group


hsa_status_t rocprofiler_group_read(
    rocprofiler_group_t* group);          // [in/out] profiling group


hsa_status_t rocprofiler_group_get_data(
    rocprofiler_group_t* group);          // [in/out] profiling group
```

8.4.5 Intercepting API

The library provides a callback API for enabling profiling for the kernels dispatched to HSA AQL queues. The API enables per-kernel profiling data collection. Currently implemented the option with serializing the kernels execution.

ROC profiler callback type:

```
hsa_status_t (*rocprofiler_callback_t)(
    const rocprofiler_callback_data_t* callback_data, // callback data passed by HSA␣
↪runtime
    void* user_data,                                  // [in/out] user data passed
                                                     // to the callback
    rocprofiler_group** group);                       // [out] returned profiling group
```

Profiling callback data:

```
typedef struct {
    uint64_t dispatch;                                // dispatch timestamp
    uint64_t begin;                                   // begin timestamp
    uint64_t end;                                     // end timestamp
    uint64_t complete;                                // completion signal timestamp
} rocprofiler_dispatch_record_t;

typedef struct {
    hsa_agent_t agent;                                // GPU agent handle
    uint32_t agent_index;                             // GPU index
    const hsa_queue_t* queue;                         // HSA queue
    uint64_t queue_index;                             // Index in the queue
    const hsa_kernel_dispatch_packet_t* packet;       // HSA dispatch packet
    const char* kernel_name;                          // Kernel name
    const rocprofiler_dispatch_record_t* record;      // Dispatch record
} rocprofiler_callback_data_t;
```

Queue callbacks:

```
typedef struct {
    rocprofiler_callback_t dispatch;                          // kernel dispatch␣
↪callback
    hsa_status_t (*destroy)(hsa_queue_t* queue, void* data);  // queue destroy␣
↪callback
} rocprofiler_queue_callbacks_t;
```

Adding/removing kernel dispatch and queue destroy callbacks:

```
hsa_status_t rocprofiler_set_intercepting(
    rocprofiler_intercepting_t callbacks,                     // intercepting␣
↪callbacks
    void* data);                                              // [in/out] passed␣
↪callbacks data

hsa_status_t rocprofiler_remove_intercepting();
```

8.4.6 Profiling context pools

The API provide capability to create a context pool for a given agent and a set of features, to fetch/release a context entry, to register a callback for pool's contexts completion. Profiling pool handle:

```
typename rocprofiler_pool_t;
Profiling pool entry:
typedef struct {
    rocprofiler_t* context;          // context object
    void* payload;                   // payload data object
} rocprofiler_pool_entry_t;
```

Profiling handler, calling on profiling completion:

```
typedef bool (*rocprofiler_pool_handler_t)(const rocprofiler_pool_entry_t* entry, void*␣
↪arg);
```

Profiling properties:

```
typedef struct {
uint32_t num_entries;                    // pool size entries
uint32_t payload_bytes;                  // payload size bytes
rocprofiler_pool_handler_t handler;      // handler on context completion
void* handler_arg;                       // the handler arg
} rocprofiler_pool_properties_t;
```

Open profiling pool:

```
hsa_status_t rocprofiler_pool_open(
hsa_agent_t agent,                       // GPU handle
rocprofiler_feature_t* features,         // [in] profiling features array
uint32_t feature_count,                  // profiling info count
rocprofiler_pool_t** pool,               // [out] context object
uint32_t mode,                           // profiling mode mask
rocprofiler_pool_properties_t*);         // pool properties
```

Close profiling pool:

```
hsa_status_t rocprofiler_pool_close(
rocprofiler_pool_t* pool);               // profiling pool handle
```

Fetch profiling pool entry:

```
hsa_status_t rocprofiler_pool_fetch(
rocprofiler_pool_t* pool,          // profiling pool handle
rocprofiler_pool_entry_t* entry);  // [out] empty profiling pool entry
```

Release profiling pool entry:

```
hsa_status_t rocprofiler_pool_release(
rocprofiler_pool_entry_t* entry);  // released profiling pool entry
```

Iterate fetched profiling pool entries:

```
hsa_status_t rocprofiler_pool_iterate(
rocprofiler_pool_t* pool,           // profiling pool handle
hsa_status_t (*callback)(rocprofiler_pool_entry_t* entry, void* data),
                                    // callback
void *data);                                // [in/out] data passed to callback
```

Flush completed entries in profiling pool:

```
hsa_status_t rocprofiler_pool_flush(
    rocprofiler_pool_t* pool);       // profiling pool handle
```

8.5 Application code examples

8.5.1 Querying available metrics

Info data callback:

```
hsa_status_t info_data_callback(const rocprofiler_info_data_t info, void *data) {
    switch (info.kind) {
        case ROCPROFILER_INFO_KIND_METRIC: {
            if (info.metric.expr != NULL) {
                fprintf(stdout, "Derived counter:  gpu-agent%d : %s : %s\n",
                    info.agent_index, info.metric.name, info.metric.description);
                fprintf(stdout, "      %s = %s\n", info.metric.name, info.metric.expr);
            } else {
                fprintf(stdout, "Basic counter:  gpu-agent%d : %s",
                    info.agent_index, info.metric.name);
                if (info.metric.instances > 1) {
                    fprintf(stdout, "[0-%u]", info.metric.instances - 1);
                }
                fprintf(stdout, " : %s\n", info.metric.description);
                fprintf(stdout, "      block %s has %u counters\n",
                    info.metric.block_name, info.metric.block_counters);
            }
            fflush(stdout);
            break;
        }
        default:
            printf("wrong info kind %u\n", kind);
            return HSA_STATUS_ERROR;
    }
    return HSA_STATUS_SUCCESS;
}
```

Printing all available metrics:

```
hsa_status_t status = rocprofiler_iterate_info(
    agent,
    ROCPROFILER_INFO_KIND_METRIC,
    info_data_callback,
```

(continues on next page)

(continued from previous page)

```
    NULL);
<check status>
```

8.5.2 Profiling code example

Profiling of L1 miss ratio, average memory bandwidth

In the example below `rocprofiler_group_get_data` group APIs are used for the purpose of a usage example but in `SINGLEGROUP` mode when only one group is allowed the context handle itself can be saved and then direct context method `rocprofiler_get_data` with default group index equal to 0 can be used.

```
hsa_status_t dispatch_callback(
    const rocprofiler_callback_data_t* callback_data,
    void* user_data,
    rocprofiler_group_t* group)
{
    hsa_status_t status = HSA_STATUS_SUCCESS;
    // Profiling context
    rocprofiler_t* context;
    // Profiling info objects
    rocprofiler_feature_t features* = new rocprofiler_feature_t[2];
    // Tracing parameters
    rocprofiler_feature_parameter_t* parameters = new rocprofiler_feature_parameter_t[2];

    // Setting profiling features
    features[0].type = ROCPROFILER_METRIC;
    features[0].name = "L1_MISS_RATIO";
    features[1].type = ROCPROFILER_METRIC;
    features[1].name = "DRAM_BANDWIDTH";

    // Creating profiling context
    status = rocprofiler_open(callback_data->dispatch.agent, features, 2, &context,
        ROCPROFILER_MODE_SINGLEGROUP, NULL);
    <check status>

    // Get the profiling group
    // For general case with many groups there is rocprofiler_group_count() API
    const uint32_t group_index = 0
    status = rocprofiler_get_group(context, group_index, group);
    <check_status>

    // In SINGLEGROUP mode the context handle itself can be saved, because there is just␣
↪one group
    <saving the callback data/profiling group/profiling features>

    return status;
}
```

Profiling tool constructor is adding the dispatch callback:

```
void profiling_libary_constructor() {
    // Defining callback data, no data in this simple example
```

(continues on next page)

(continued from previous page)

```
    void* callback_data = NULL;

    // Adding observers
    hsa_sttaus_t status = rocprofiler_add_dispatch_callback(dispatch_callback, callback_
↪data);
    <check status>

    // Dispatching profiled kernel
    <dispatching profiled kernels>
}

void profiling_libary_destructor() {
    <for entry : <saved callbacks data>> {
        // In SINGLEGROUP mode the rocprofiler_get_group() method with default zero group
        // index can be used, if context handle would be saved
        status = rocprofiler_group_get_data(entry->group);
        <check status>
        status = rocprofiler_get_metrics(entry->group->context);
        <check status>
        status = rocprofiler_close(entry->group->context);
        <check status>

        <tool_dump_data_method(entry->dispatch_data, entry->features, entry->features_
↪count)>;
    }
}
```

8.5.3 Option to use completion callback

Creating profiling context with completion callback:

```
...
rocprofiler_properties_t properties = {};
properties.callback = completion_callback;
properties.callback_arg = NULL;     // no args defined
status = rocprofiler_open(agent, features, 3, &context,
    ROCPROFILER_MODE_SINGLEGROUP, properties);
<check status>
...
```

Definition of completion callback:

```
void completion_callback(profiler_group_t group, void* arg) {
    <tool_dump_data_method(group)>
    hsa_status_t status = rocprofiler_close(group.context);
    <check status>
}
```

8.5.4 Option to use context pool

Code example of context pool usage.

Creating profiling contexts pool:

```
 ...
rocprofiler_pool_properties_t properties{};
properties.num_entries = 100;
properties.payload_bytes = sizeof(context_entry_t);
properties.handler = context_handler;
properties.handler_arg = handler_arg;
status = rocprofiler_pool_open(agent, features, 3, &context,
             ROCPROFILER_MODE_SINGLEGROUP, properties);
<check status>
 ...
```

Fetching a context entry:

```
 rocprofiler_pool_entry_t pool_entry{};
status = rocprofiler_pool_fetch(pool, &pool_entry);
<check status>
// Profiling context entry
rocprofiler_t* context = pool_entry.context;
context_entry_t* entry = reinterpret_cast <context_entry_t*>
                         (pool_entry.payload);
```

8.5.5 Standalone sampling usage code example

The profiling metrics are being read from separate standalone queue other than the application kernels are submitted to. To enable the sampling mode, the profiling mode in all user queues should be enabled. It can be done by loading ROC-profiler library to HSA runtime using the environment variable `HSA_TOOLS_LIB` for all shell sessions.

```
 // Sampling rate
uint32_t sampling_rate = <some rate>;
// Sampling count
uint32_t sampling_count = <some count>;
// HSA status
hsa_status_t status = HSA_STATUS_ERROR;
// HSA agent
hsa_agent_t agent;
// Profiling context
rocprofiler_t* context = NULL;
// Profiling properties
rocprofiler_properties_t properties;

// Getting HSA agent
<query for HSA agent by 'hsa_iterate_agents()'>

// Profiling feature objects
const unsigned feature_count = 2;
rocprofiler_feature_t feature[feature_count];
```

(continues on next page)

(continued from previous page)

```
// Counters and metrics
feature[0].kind = ROCPROFILER_FEATURE_KIND_METRIC;
feature[0].name = "GPUBusy";
feature[1].kind = ROCPROFILER_FEATURE_KIND_METRIC;
feature[1].name = "SQ_WAVES";

// Creating profiling context with standalone queue
properties = {};
properties.queue_depth = 128;
status = rocprofiler_open(agent, feature, feature_count, &context,
         ROCPROFILER_MODE_STANDALONE| ROCPROFILER_MODE_CREATEQUEUE|
         ROCPROFILER_MODE_SINGLEGROUP, &properties);
<check status>

// Start counters and sample them in the loop with the sampling rate
status = rocprofiler_start(context, 0);
<check status>

for (unsigned ind = 0; ind < sampling_count; ++ind) {
   sleep(sampling_rate);
   status = rocprofiler_read(context, 0);
   <check status>
   status = rocprofiler_get_data(context, 0);
   <check status>
   status = rocprofiler_get_metrics(context);
   <check status>
   print_results(feature, feature_count);
}

// Stop counters
status = rocprofiler_stop(context, group_n);
<check status>

// Finishing cleanup
// Deleting profiling context will delete all allocated resources
status = rocprofiler_close(context);
<check status>
```

8.5.6 Printing out profiling results

The following is a code example for printing out the profiling results from profiling features array:

```
void print_results(rocprofiler_feature_t* feature, uint32_t feature_count) {
for (rocprofiler_feature_t* p = feature; p < feature + feature_count; ++p)
{
    std::cout << (p - feature) << ": " << p->name;
    switch (p->data.kind) {
        case ROCPROFILER_DATA_KIND_INT64:
            std::cout << " result_int64 (" << p->data.result_int64 << ")"
                    << std::endl;
            break;
```

(continues on next page)

(continued from previous page)

```
        case ROCPROFILER_DATA_KIND_BYTES: {
            std::cout << " result_bytes ptr(" << p->data.result_bytes.ptr <<
                ") " << " size(" << p->data.result_bytes.size << ")"
                << " instance_count(" << p->data.result_bytes.instance_count
                << ")";
            break;
        }
        default:
            std::cout << "bad result kind (" << p->data.kind << ")"
                    << std::endl;
            <abort>
    }
}
}
```

CHAPTER

NINE

ROCPROFILER

9.1 Modules

9.2 Data Structures

9.2.1 Data Structures

9.2.2 Data Structure Index

9.2.3 Data Fields

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

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

9.3 Files

9.3.1 File List

9.3.2 Globals

All

Globals

Globals

Globals

Globals

Globals

Globals

Globals

CHAPTER

TEN

ROCPROFILERV2 API

ROCProfilerV2 provides an API that allows fine-grained control over the profiling process. Like the CLI tool `rocprofv2`, the API supports both application tracing and kernel profiling. ROCProfilerV2 API allows you to create a session and invoke application tracing or kernel profiling within the session. The following sections describe session management, application tracing and kernel profiling using ROCProfilerV2 API.

10.1 Profiling sessions

A ROCProfilerV2 session maintains the global profiling state for an application. It is a unique identifier for a profiling or tracing task that is specified within the session. A session contains sufficient information about what needs to be collected or traced and it allows you to start/stop profiling/tracing as and when required.

The following demonstrates the use of session management APIs:

```
// Initialize the tools
   rocprofiler_initialize();

// Create the session with no replay mode
   rocprofiler_session_id_t session_id;
   rocprofiler_create_session(ROCPROFILER_NONE_REPLAY_MODE, &session_id);

// Start Session
   rocprofiler_start_session(session_id);

// profile a kernel -kernelA
   hipLaunchKernelGGL(kernelA, dim3(1), dim3(1), 0, 0);

// Deactivating session
   rocprofiler_terminate_session(session_id);

// Destroy sessions
   rocprofiler_destroy_session(session_id);

// Destroy all profiling related objects
   rocprofiler_finalize();
```

The following is a typical session management workflow:

1. Initialize ROCProfilerV2 using `rocprofiler_initialize`.
2. Create a session using `rocprofiler_create_session`. The created session keeps track of the global status of the application profiling.

Note: You can only create a session in no-replay mode (ROCPROFILER_NONE_REPLAY_MODE) which allows kernels to be run only once.

3. Create a buffer to hold the results using `rocprofiler_create_buffer`.
4. Create filters using `rocprofiler_create_filter` to specify the profiling task such as application tracing or metrics/counters collection.

Note: If the same filter is applied twice with different values, the latter application of the filter is considered the recent one, which overwrites the former application of the filter. To learn about the types of filters and their utility, see *Filters*.

5. Start the session with `rocprofiler_start_session`.
6. Run the specified kernels to collect traces or counters/metrics (as specified in the filter)
7. Terminate the session with `rocprofiler_terminate_session` and flush the profiling results using `rocprofiler_flush_data`.

Note: The session must be terminated after the kernel completes (synchronization required). If a session is stopped before the completion of kernel execution within that session, the instrumentation data is undefined. Additionally, a session can be restarted after terminating.

8. Destroy the session with `rocprofiler_destroy_session` and finalize profiling with `rocprofiler_finalize`.

See working examples demonstrating the use of the ROCProfilerV2 API in *Application Tracing* and *Kernel Profiling*.

10.2 Filters

As explained in *Profiling Sessions*, filters allow you to specify a profiling task within a session. For different profiling tasks, different filters are specified as a parameter to `rocprofiler_create_filter`.

See the list of filters in the table below:

Filter	Purpose
`ROCPROFILER_API_TRACE`	To trace API calls. You must specify the API calls to be traced, in a vector.
`ROCPROFILER_DISPATCH _TIMESTAMPS_COLLECTION`	To track all the kernel execution's start and end times on the GPUs
`ROCPROFILER_COUNTERS_COLLECTION`	To collect counters. You must specify the counters to be collected, in a vector.

10.3 Application tracing

The following code demonstrates the usage of ROCProfilerV2 APIs to trace an application. This example traces HIP APIs, HIP asynchronous activities, HSA APIs, HSA asynchronous activities, and ROCTX ranges. Note the use of ROCPROFILER_API_TRACE filter to trace API calls, and ROCPROFILER_DISPATCH_TIMESTAMPS_COLLECTION filter to trace the kernel.

```
int main(int argc, char*\* argv) {
   int\* gpuMem;
   prepare();
   // Initialize the tools
   CHECK_ROCPROFILER(rocprofiler_initialize());

   // Creating the session with given replay mode
   rocprofiler_session_id_t session_id;
   CHECK_ROCPROFILER(rocprofiler_create_session(ROCPROFILER_NONE_REPLAY_MODE, &session_
↪id));

   // Creating Output Buffer for the data
   rocprofiler_buffer_id_t buffer_id;
   CHECK_ROCPROFILER(rocprofiler_create_buffer(session_id,
      [](const rocprofiler_record_header_t\* record, const
      rocprofiler_record_header_t\* end_record,
      rocprofiler_session_id_t session_id, rocprofiler_buffer_id_t
      buffer_id) {
         WriteBufferRecords(record, end_record, session_id, buffer_id);
      },
      0x9999, &buffer_id));

   // Specifying the APIs to be traced in a vector
   std::vector<rocprofiler_tracer_activity_domain_t> apis_requested;
   apis_requested.emplace_back(ACTIVITY_DOMAIN_HIP_API);
   apis_requested.emplace_back(ACTIVITY_DOMAIN_HIP_OPS);
   apis_requested.emplace_back(ACTIVITY_DOMAIN_HSA_API);
   apis_requested.emplace_back(ACTIVITY_DOMAIN_HSA_OPS);
   apis_requested.emplace_back(ACTIVITY_DOMAIN_ROCTX);
   rocprofiler_filter_id_t api_tracing_filter_id;

   // Creating filter for tracing APIs
   CHECK_ROCPROFILER(rocprofiler_create_filter(
      session_id, ROCPROFILER_API_TRACE,
      rocprofiler_filter_data_t{&apis_requested[0]}, apis_requested.size(),
      &api_tracing_filter_id, rocprofiler_filter_property_t{}));
   CHECK_ROCPROFILER(rocprofiler_set_filter_buffer(session_id,
      api_tracing_filter_id, buffer_id));

   // Kernel Tracing
   rocprofiler_filter_id_t kernel_tracing_filter_id;
   CHECK_ROCPROFILER(rocprofiler_create_filter(session_id,
      ROCPROFILER_DISPATCH_TIMESTAMPS_COLLECTION, rocprofiler_filter_data_t{},
      0, &kernel_tracing_filter_id, rocprofiler_filter_property_t{}));
   CHECK_ROCPROFILER(rocprofiler_set_filter_buffer(session_id,
      kernel_tracing_filter_id, buffer_id));
```

(continues on next page)

(continued from previous page)

```
    // Normal HIP Calls won't be traced
    hipDeviceProp_t devProp;
    HIP_CALL(hipGetDeviceProperties(&devProp, 0));
    HIP_CALL(hipMalloc((void**)&gpuMem, 1 * sizeof(int)));
    // KernelA and KernelB won't be traced
    kernelCalls('A');
    kernelCalls('B');

    // Activating Profiling Session to profile whatever kernel launches occur
    // up to the next terminate session
    CHECK_ROCPROFILER(rocprofiler_start_session(session_id));

    // KernelC, KernelD, KernelE and KernelF to be traced as part of the session
    kernelCalls('C');
    kernelCalls('D');
    kernelCalls('E');
    kernelCalls('F');
    // Normal HIP Calls that will be traced
    HIP_CALL(hipFree(gpuMem));

    // Deactivating session
    CHECK_ROCPROFILER(rocprofiler_terminate_session(session_id));

    // Manual Flush user buffer request
    CHECK_ROCPROFILER(rocprofiler_flush_data(session_id, buffer_id));

    // Destroy sessions
    CHECK_ROCPROFILER(rocprofiler_destroy_session(session_id));

    // Destroy all profiling related objects (User buffer, sessions, filters, etc..)
    CHECK_ROCPROFILER(rocprofiler_finalize());

    return 0;
}
```

10.4 Kernel profiling

The following is a full-application example that utilizes the ROCProfilerV2 API to profile the kernels. The `ROCPROFILER_COUNTERS_COLLECTION` filter for counter collection distinguishes this example from the one in *Application tracing*. The `GRBM_COUNT` counter to be collected is specified in a vector of strings as shown.

```
#include <hip/hip_runtime.h>
#include <rocprofiler/v2/rocprofiler.h>

int main(int argc, char*\* argv) {
    int\* gpuMem;

    // Initialize the tools
    CHECK_ROCPROFILER(rocprofiler_initialize());
```

(continues on next page)

(continued from previous page)

```
// Creating the session with given replay mode
rocprofiler_session_id_t session_id;
CHECK_ROCPROFILER(rocprofiler_create_session(ROCPROFILER_NONE_REPLAY_MODE,
&session_id));

// Creating Output Buffer for the data
rocprofiler_buffer_id_t buffer_id;
CHECK_ROCPROFILER(rocprofiler_create_buffer(session_id,
   [](const rocprofiler_record_header_t* record, const
   rocprofiler_record_header_t* end_record,
   rocprofiler_session_id_t session_id, rocprofiler_buffer_id_t
   buffer_id) {
      WriteBufferRecords(record, end_record, session_id, buffer_id);
   },
   0x9999, &buffer_id));

// Counter Collection Filter
std::vector<const char*> counters;
counters.emplace_back("GRBM_COUNT");
rocprofiler_filter_id_t filter_id;
[[maybe_unused]] rocprofiler_filter_property_t property = {};
CHECK_ROCPROFILER(rocprofiler_create_filter(session_id,
   ROCPROFILER_COUNTERS_COLLECTION,
   rocprofiler_filter_data_t{.counters_names = &counters[0]},
   counters.size(), &filter_id, property));
CHECK_ROCPROFILER(rocprofiler_set_filter_buffer(session_id, filter_id,
   buffer_id));

// Normal HIP Calls
hipDeviceProp_t devProp;
HIP_CALL(hipGetDeviceProperties(&devProp, 0));
HIP_CALL(hipMalloc((void**)&gpuMem, 1 * sizeof(int)));

// KernelA and KernelB won't be profiled
kernelCalls('A');
kernelCalls('B');

// Activating Profiling Session to profile whatever kernel launches occur
// up to the next terminate session
CHECK_ROCPROFILER(rocprofiler_start_session(session_id));

// KernelC, KernelD, KernelE and KernelF to be profiled as part of the session
kernelCalls('C');
kernelCalls('D');
kernelCalls('E');
kernelCalls('F');
// Normal HIP Calls
HIP_CALL(hipFree(gpuMem));

// Deactivating session
CHECK_ROCPROFILER(rocprofiler_terminate_session(session_id));
```

(continues on next page)

(continued from previous page)

```
    // Manual Flush user buffer request
    CHECK_ROCPROFILER(rocprofiler_flush_data(session_id, buffer_id));

    // Destroy sessions
    CHECK_ROCPROFILER(rocprofiler_destroy_session(session_id));

    // Destroy all profiling related objects (User buffer, sessions, filters, etc..)
    CHECK_ROCPROFILER(rocprofiler_finalize());

    return 0;
}
```

CHAPTER

ELEVEN

LICENSE