**ROC Profiler Library Specification**

rev 1.2.7

© Advanced Micro Devices Corp. 2018

All rights reserved.

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Authors** | **Description** |
| 1.0.0 | 08/31/2017 | Evgeny Shcherbakov | Initial version |
| 1.0.1 | 09/01/2017 | Evgeny Shcherbakov | Adding metric object type declaration and other changes |
| 1.1.0 | 09/18/2017 | Evgeny Shcherbakov | Renaming to rocprofiler library, context/info APIs, observer new/delete/simple APIs, code examples |
| 1.1.1 | 09/19/2017 | Evgeny Shcherbakov | Minor fixes of the code example 6.1 and others |
| 1.1.2 | 09/28/2017 | Evgeny Shcherbakov | Minor fix |
| 1.2.0 | 10/30/2017 | Evgeny Shcherbakov | Update, introducing explicit groups usage, GPU index instead of agent handle, context callback API |
| 1.2.1 | 11/03/2017 | Evgeny Shcherbakov | Minor fixes |
| 1.2.2 | 11/09/2017 | Evgeny Shcherbakov | Code examples fixes |
| 1.2.3 | 11/22/2017 | Evgeny Shcherbakov | Renamings; gpu index changed on HSA agent; info iterate/query API |
| 1.2.4 | 12/13/2017 | Evgeny Shcherbakov | Minor updates and typos. More details on profiling mode, properties, and groups. |
| 1.2.5 | 01/05/2018 | Evgeny Shcherbakov | Library major/minor version; Info API changes, added agent parameter, \_info\_query API changed |
| 1.2.6 | 04/24/2018 | Evgeny Shcherbakov | Callback and activity APIs |
| 1.2.7 | 04/30/2018 | Evgeny Shcherbakov | Activity records iterate method; the code example fixes |

**Contents**

[Revision history 2](#_Toc512818094)

[1. High level overview 3](#_Toc512818095)

[2. Library implementation requirements 3](#_Toc512818096)

[3. HSA runtime requirements 4](#_Toc512818097)

[4. Library basic usage flow 5](#_Toc512818098)

[4.1. Per kernel data collection, profiling in the application context 5](#_Toc512818099)

[4.2. Statistical profiling / sampling, in the tool context, external to application context. 5](#_Toc512818100)

[4.3. Environment 5](#_Toc512818101)

[5. General API 5](#_Toc512818102)

[5.1. Description 5](#_Toc512818103)

[5.2. Returning the error and error string methods 6](#_Toc512818104)

[5.3. Library version 6](#_Toc512818105)

[6. Backend API 6](#_Toc512818106)

[6.1. Description 6](#_Toc512818107)

[6.2. Info API 7](#_Toc512818108)

[6.3. Context API 8](#_Toc512818109)

[6.4. Sampling API 11](#_Toc512818110)

[6.5. Intercepting API 12](#_Toc512818111)

[7. Frontend API 12](#_Toc512818112)

[7.1. Description 12](#_Toc512818113)

[7.2. Runtime API domains 13](#_Toc512818114)

[7.3. Callback API 14](#_Toc512818115)

[7.4. Activity API 15](#_Toc512818116)

[8. Application code examples 16](#_Toc512818117)

[8.1. Querying available metrics, traces, SQTT trace parameters 16](#_Toc512818118)

[8.2. Profiling code example 17](#_Toc512818119)

[8.3. Option to use completion callback 18](#_Toc512818120)

[8.4. Callback API 19](#_Toc512818121)

[8.5. Activity API 19](#_Toc512818122)

[8.6. Runtime API tracing 20](#_Toc512818123)

# **High level overview**

The goal of the implementation is to provide a HW specific low-level performance analysis interface for profiling of GPU compute applications. The profiling includes HW performance counters (PMC) with complex performance metrics, thread trace (SQTT) and streaming performance counters (SPM). The implementation distinguishes two profiling features, metrics and traces. HW performance counters are treated as the basic metrics and the formulas can be defined for derived complex metrics. SQTT and SPM are examples of HW traces.

The library can be loaded by HSA runtime as a tool plugin and it can be loaded by higher level HW independent performance analysis API like PAPI.

The library has C API and is based on AQLprofile AMD specific HSA extension.

# **Library implementation requirements**

* 1. The library provides profiling based on the following HW profiling features:
     + Performance counters in accumulating mode (PMC)
     + SQ Thread Trace (SQTT)
     + Streaming Performance Monitors (SPM)
  2. The library provides methods to query the list of supported HW features.
  3. The library provides profiling APIs to start, stop, read metrics results and tracing data.
  4. The library provides a callback API for collecting per-kernel profiling data for the kernels dispatched to HSA AQL queues.
  5. The library provides mechanism to load profiling tool library plugin by env variable ROCP\_TOOL\_LIB.
  6. The library is responsible for allocation of the buffers for profiling and notifying about output data buffer overflow for traces.
  7. The library is implemented based on AMD specific AQLprofile HSA extension.
  8. The library implementation is abstracted from the specific GFXIP.
  9. The library implementation is extensible:
* Easy adding of counters and metrics
* Counters enumeration
* Counters and metrics can be dynamically configured using XML configuration files 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
* Easy extending of information provided to the profiling tool by extending the ROC profiler dispatch callback data.

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>

. . .

</gfx9>

<global>

<metric name=L1\_MISS\_RATIO expr=L1\_CYCLES\_COUNT/ L1\_MISS\_COUNTER ></metric>

</global>

Or another variant I’m thinking about:

<gfx8>

<L1>

<event name=CYCLES id=0 >

<event name=MISS event=33 >

. . .

</L1>

. . .

</gfx8>

<global>

<metric name=L1\_MISS\_RATIO expr=100\*L1.MISS/ L1.CYCLES ></metric>

</global>

# **HSA runtime requirements**

* 1. HSA runtime should support mechanism to intercept dispatched AQL packets and to return modified packets sequence.

# **Library basic usage flow**

## Per kernel data collection, profiling in the application context

ROC Profiler library is loaded by HSA runtime. ROC profiler is intercepting the HSA AQL queue create API and returns a proxy queue which provides a dispatch callback interface for the profiler.

HSA application

Profiling Tool

HSA ProxyQueue interface

HSA runtime

ROC Profiler lib

loaded by HSA runtime

## Statistical profiling / sampling, in the tool context, external to application context.

AQL Profile sampling API

HW counters start/read/stop

Profiling tool context

HSA AQL Queue interface

HW independent API

PAPI Preset events - predefined events, are a common set of events deemed relevant and useful for application performance tuning.

Performance API

(PAPI)

ROC Profiler lib

Profiling Tool

(TAU/Eclipse PTP)

HSA runtime

High-Level Language

runtime

HSA application context

HSA runtime

## Environment

* 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 ROC Profiler
* ROCP\_HSA\_INTERCEPT – if set then HSA dispatches intercepting is enabled

# **General API**

## Description

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

Returning the error and error string methods:

* rocprofiler\_errno – method for returning the error number
* 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

## Returning the error and error string methods

rocprofiler\_errno\_t rocprofiler\_errno();

const char\* rocprofiler\_error\_string();

## 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 venison:

uint32\_t rocprofiler\_major\_version();

uint32\_t rocprofiler\_minor\_version();

# **Backend API**

## Description

The library provides the methods to open/close profiling context, to start, stop and read HW performance counters PMC and thread traces SQTT, to intercept kernel dispatches to collect per-kernel profiling data. Also the library 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.

For collecting per-kernel data for the submitted to HSA queues kernels 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

Info API:

* rocprofiler\_info\_kind\_t – profiling info kind
* rocprofiler\_info\_query\_t – profiling info query
* rocprofiler\_info\_data\_t – profiling info data
* rocprofiler\_iterate\_info – 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

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\_set\_intercepting – adding kernel dispatch and queue destroy callbacks
* rocprofiler\_remove\_intercepting – removing intercepting callbacks

## Info API

The profiling metrics are defined by name and the traces are defined by name and parameters. All supported features and parameters 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 same as in the hardware documentation and the parameters codes are rocprofiler\_feature\_parameter\_t values, see below in the ‘Context API’ section. The trace names can be “SQTT” for SQ thread trace and “SPM” for streaming performance monitors.

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\_TRACE\_PARAMETER = 4 // trace parameter info

} rocprofiler\_info\_kind\_t;

Profiling info data:

typedef struct {

rocprofiler\_info\_kind\_t kind; // info data kind

union {

struct {

const char\* name; // metric name

const char\* description; // metric description

} metric;

struct {

const char\* name; // trace name

const char\* description; // trace description

uint32\_t parameter\_count; // supported by the trace number parameters

} trace;

struct {

uint32\_t code; // parameter code

const char\* trace\_name; // trace name

const char\* parameter\_name; // parameter name

const char\* description; // trace parameter description

} trace\_parameter;

};

} rocprofiler\_info\_data\_t;

Return info for a given info kind:

has\_status\_t rocprofiler\_get\_info(

hsa\_agent\_t agent, // GPU handle

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(

hsa\_agent\_t agent, // GPU handle

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(

hsa\_agent\_t agent, // GPU handle

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

## 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 it is required more than one run to collect all requested counters data then 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 responsibility of the tool. The groups are automatically identified on the profiling context open and there is API to access them, see the ‘Profiling groups’ section below.

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\_TIMESTAMP = 2 // timestamp

} 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 also specified. See also ‘Profiler properties’ section below 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

// HSA 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’ 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

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 per 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 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 instance, 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

## 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

## 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.

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

ROC profiler callback data type:

typedef struct {

struct {

hsa\_agent\_t agent; // GPU device

const char\* kernel\_name; // dispatched kernel mangled name,

// depends on specific high-level language runtime

uint64\_t queue\_index; // queue write index

} dispatch;

} rocprofiler\_callback\_data\_t;

Adding/removing kernel dispatch and queue remove callbacks

typedef struct {

rocprofiler\_callback\_t dispatch; // kernel dispatch callback

hsa\_status\_t (\*destroy)(hsa\_queue\_t\* queue, void\* data); // queue destroy callback

} rocprofiler\_intercepting\_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();

# **Frontend API**

## Description

ROC profiler frontend provides support for runtime API callbacks and activity records logging. The APIs of the different runtimes of at the different runtime levels, for example language level and driver level, are considered as different API domains with assigned domain IDs. The API callbacks provide the API calls arguments and are called on different phases, on enter, on exit, on kernel completion. The activity records are logged to the ring buffer and can be associated with the respective API callbacks using the correlation ID. Activity API can be used to enable collecting of the records with timestamping data for API calls and the kernel submits.

Methods return non-zero on error and library errno is set.

Runtime API domains:

* rocprofiler\_api\_domain\_t – runtime API domains, HIP, OpenCL, HAS, etc…
* rocprofiler\_activity\_class\_t - enumeration of runtime API domains activity classes
* rocprofiler\_<domain>\_api\_cid\_t – runtime API calls ID enum. HIP API, OpenCL API, etc…
* rocprofiler\_get\_api\_name – return method name by given API domain and call id

Callback API:

* rocprofiler\_api\_phase\_t – runtime API pahse enum, on enter, on exit, on complete
* rocprofiler\_<domain>\_api\_callback\_data\_t - runtime API callback data type
* rocprofiler\_api\_callback\_t – runtime API callback type
* rocprofiler\_enable\_api\_callback – enable runtime API callback
* rocprofiler\_disable\_api\_callback – disable runtime API callback

Actiovity API:

* rocprofiler\_activity\_record\_t – activity record
* rocprofiler\_activity\_record\_callback\_t – activity record callback
* rocprofiler\_activity\_iterate\_record – iterating activity records by using the callback
* rocprofiler\_activity\_enable – enable activity records logging
* rocprofiler\_activity\_disable – disable activity records logging
* rocprofiler\_activity\_set\_properties – set activity properties, ring buffer size, buffer full callback

## Runtime API domains

The APIs of the different runtimes of at the different runtime levels, for example language level and driver level, are considered as different API domains with assigned domain IDs.

Enumeration of runtime API domains ID for HSA, HIP, HCC, OpenCL, OpenMP, VDI (Virtual Device interface) layer, etc…:

typedef enum {

ROCPROFILER\_API\_DOMAIN\_ALL,

ROCPROFILER\_API\_DOMAIN\_HSA,

ROCPROFILER\_API\_DOMAIN\_HIP,

ROCPROFILER\_API\_DOMAIN\_HCC,

ROCPROFILER\_API\_DOMAIN\_OCL,

ROCPROFILER\_API\_DOMAIN\_OMP,

ROCPROFILER\_API\_DOMAIN\_VDI,

} rocprofiler\_api\_domain\_t;

Enumeration of runtime domains activity classes:

typedef enum {

ROCPROFILER\_ACTIVITY\_CLASS\_API,

. . .

} rocprofiler\_activity\_class\_t;

Enumeration of runtime API calls ID, an example for HIP:

typedef enum {

ROCPROFILER\_HIP\_API\_ID\_ALL,

ROCPROFILER\_HIP\_API\_ID\_hipMalloc,

ROCPROFILER\_HIP\_API\_ID\_hipFree,

ROCPROFILER\_HIP\_API\_ID\_hipMemcpy,

ROCPROFILER\_HIP\_API\_ID\_hipLaunchKernel,

. . .

} rocprofiler\_hip\_api\_cid\_t;

Return method name by given API domain and call id:

const char\* rocprofiler\_get\_api\_name( // NULL returned on error and error number is set

rocprofiler\_api\_domain\_t domain, // API domain

rocprofiler\_hip\_api\_cid\_t cid // API call id

};

## Callback API

ROC profiler frontend provides support for runtime API callbacks and activity records logging. The API callbacks provide the API calls arguments and are called on different phases, on enter, on exit, on kernel completion.

Methods return non-zero on error and library errno is set.

API phase passed to the callbacks:

typedef enum {

ROCPROFILER\_API\_PHASE\_ENTER,

ROCPROFILER\_API\_PHASE\_EXIT,

ROCPROFILER\_API\_PHASE\_COMPLETE,

} rocprofiler\_api\_phase\_t;

Runtime API callback data type:

typedef struct {

rocprofiler\_correletion\_id\_t correlation\_id;

union {

struct {

void\* ptr;

size\_t sizeBytes;

} hipMalloc;

struct {

void\* ptr;

} hipFree;

. . .

} args;

} rocprofiler\_hip\_api\_callback\_t;

Runtime API callback type:

typedef void (\*rocprofiler\_api\_callback\_t)(

rocprofiler\_api\_domain\_t domain. // runtime API domain

uint32\_t cid, // API call ID

const void\* callback\_data, // [in] callback data with correlation id and

// the call arguments

void\* user\_data); // [in/out] user passed data

Enable/disable runtime API callbacks:

int rocprofiler\_enable\_api\_callback(

rocprofiler\_api\_domain\_t domain. // runtime API domain

uint32\_t cid, // API call ID

void\* callback, // [in] callback function pointer

void\* user\_data); // [in/out] user passed data

int rocprofiler\_disable\_api\_callback(

rocprofiler\_api\_domain\_t domain, // runtime API domain

uint32\_t cid); // API call ID

## Activity API

The activity records are asynchronously logged to the buffers pool and can be associated with the respective API callbacks using the correlation ID. Activity API can be used to enable collecting of the records with timestamping data for API calls and the kernel submits.

Methods return non zero on error and the library errno is set.

Activity record:

typedef struct {

rocprofiler\_api\_domain\_t domain;

rocprofiler\_activity\_class\_t activity\_class;

uint32\_t activity\_kind;

rocprofiler\_correletion\_id\_t correlation\_id; // activity correlation ID

uint64\_t begin\_ns; // begin timestamp, nano-seconds

uint64\_t end\_ns; // end timestamp, nano-seconds

union {

<activity kind specific data>

} data;

} rocprofiler\_activity\_record\_t;

Activity record callback:

typedef int (\*rocprofiler\_activity\_record\_callback\_t)(

const rocprofiler\_activity\_record\_t\* record, // Activity record

void\* arg); // passed user data

Activity records iterating using the callback:

int rocprofiler\_activity\_iterate\_records(

void\* buffer, // activity buffer pointer

uint32\_t size, // activity buffer size

rocprofiler\_activity\_record\_callback\_t record, // record callback

void\* arg); // [in/out] passed user data

Enable/disable activity records logging:

int rocprofiler\_activity\_enable(

rocprofiler\_api\_domain\_t domain, // runtime API domain

rocprofiler\_activity\_class\_t activity\_class, // activity class

uint32\_t activity\_kind); // activity kind

int rocprofiler\_activity\_disable(

rocprofiler\_api\_domain\_t domain, // runtime API domain

rocprofiler\_activity\_class\_t activity\_class, // activity class

uint32\_t activity\_kind); // activity kind

Set activity properties, ring buffer sizes and buffer full callbacks. If pool callback is NULL then the producer thread will wait until a buffer will be available:

typedef struct {

uint32\_t buffer\_size; // a buffer size in bytes

uint32\_t pool\_size; // size of pool in bytes

void (\*buffer\_cb\_fun)(void\* buffer, uint32\_t size, void\* arg); // buffer full callback

void\* buffer\_cb\_arg; // arg passed to buffer callback

void (\*pool\_cb\_fun)(void\* buffer, uint32\_t size, void\* arg); // pool full callback

void\* pool\_cb\_arg; // arg passed to pool callback

} rocprofiler\_activity\_properties\_t;

int rocprofiler\_activity\_set\_properties(

rocprofiler\_activity\_properties\_t properties); // activity logging properties

# **Application code examples**

## Querying available metrics, traces, SQTT trace parameters

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: {

printf(“metric %s, description %s\n”,

info.metric.name,

info.metric.description);

break;

}

case ROCPROFILER\_INFO\_KIND\_TRACE: {

printf(“trace %s, parameter\_count %u, description %s\n”,

info.trace.name,

info.trace.parameter\_count,

info.trace.description);

break;

}

case ROCPROFILER\_INFO\_KIND\_TRACE\_PARAMETER {

printf(“trace %s, parameter %s, code 0x%x, description %s\n”,

info.trace\_parameter.trace\_name,

info.trace\_parameter.parameter\_name,

info.trace\_parameter.parameter\_count,

info.trace\_parameter.description);

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,*

*NULL);*

*<check status>*

Printing all available traces:

hsa\_status\_t status = rocprofiler\_iterate\_info(

agent,

ROCPROFILER\_INFO\_KIND\_TRACE,

info\_data\_*callback,*

*NULL);*

*<check status>*

Printing all available SQTT trace parameters:

rocprofiler\_info\_query\_t query{};

query.info\_kind = ROCPROFILER\_INFO\_KIND\_TRACE;

query.trace.name = “SQTT”;

hsa\_status\_t status = rocprofiler\_query\_info(

agent,

ROCPROFILER\_INFO\_KIND\_TRACE\_PARAMETER

query,

info\_data\_*callback,*

*NULL);*

*<check status>*

## Profiling code example

Profiling of L1 miss ratio, average memory bandwidth and thread trace collecting.

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[3];

// 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”;

features[2].type = ROCPROFILER\_TRACE;

features[2].name = “SQTT”;

features[2].parameters = parameters;

features[2].parameter\_count = 2;

parameters[0].name = HSA\_VEN\_AMD\_AQLPROFILE\_PARAMETER\_NAME\_MASK ;

parameters[0].value = <some trace tokens mask>;

parameters[1].name = HSA\_VEN\_AMD\_AQLPROFILE\_PARAMETER\_NAME\_TOKEN\_MASK;

parameters[1].value = <some trace instructions mask>;

// Creating profiling context

status = rocprofiler\_open(callback\_data->dispatch.agent, features, 3, &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;

}

void profiling\_libary\_constructor() {

// Defining callback data, no data in this simple example

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)>;

}

}

## 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>

}

## Callback API

Callback API is used to register callbacks to be called on enter and return from the API.

void api\_callback(api\_callback\_data\_t\* data, void\* args) {

std::map<correlation\_id\_t, std::string>\* correlation\_dict\_ptr =

(std::map<correlation\_id\_t, std::string> correlation\_dict\*)args;

if (data->phase == ROCPROFILER\_CB\_PHASE\_ENTER) {

if (data->api\_id == ROCPROFILER\_API\_ID\_HIP\_KERNEL) {

(\*correlation\_dict\_ptr)[data->correlation\_id] = strdup(data->kernel->name);

} else {

(\*correlation\_dict\_ptr)[data->correlation\_id] = strdup(kAnnotationGlobal);

}

}

}

void enable\_api\_callback(std::map<correlation\_id\_t, std::string>\* correlation\_dict\_ptr) {

int success = rocprofiler\_register\_callback(ROCPROFILER\_API\_DOMAIN\_HIP,

ROCPROFILER\_API\_ID\_HIP\_KERNEL,

api\_callback, &correlation\_dict\_ptr);

<check success>

int success = rocprofiler\_register\_callback(ROCPROFILER\_API\_DOMAIN\_HIP,

ROCPROFILER\_API\_ID\_HIP\_MEMORY,

api\_callback, &correlation\_dict\_ptr);

<check success>

}

## Activity API

Activity API is used to enable collecting of the records with timestamping data about API calls and kernels submits.

Int record\_callback(rocprofiler\_activity\_record\_t\* record, void\* arg) {

std::list<rocprofiler\_activity\_record\_t>\* records\_list\_ptr = (std::list<rocprofiler\_activity\_record\_t>\*)args

records\_list\_ptr->push\_back(\*record\_ptr);

}

void activity\_complete\_callback(activity\_complete\_callback\_data\_t\* data, size\_t size, void\* arg) {

int success = rocprofiler\_activity\_iterate\_records(data, size, record\_callback, arg);

<check success>

}

void enable\_activity(std::list<rocprofiler\_activity\_record\_t>\* records\_list\_ptr) {

int success;

success = rocprofiler\_activity\_set\_ptoperties(ring\_buffer\_chunk, ring\_buffer\_size,

activity\_complete\_callback, records\_list\_ptr,

NULL, NULL);

success = rocprofiler\_activity\_enable(ROCPROFILER\_API\_DOMAIN\_HIP,

ROCPROFILER\_ACTIVITY\_CLASS\_API,

ROCPROFILER\_API\_ID\_HIP\_KERNEL);

<check success>

success = rocprofiler\_activity\_enable(ROCPROFILER\_API\_DOMAIN\_HIP,

ROCPROFILER\_ACTIVITY\_CLASS\_API,

ROCPROFILER\_API\_ID\_HIP\_MEMORY);

<check success>

}

## Runtime API tracing

Runtime API tracing is implemented using both callback and activity profiler APIs. Similar to Tensorflow internal ‘device tracer’.

std::map<correlation\_id\_t, std::string> correlation\_dict;

rocprofiler\_activity\_record\_t> records\_list;

void enable\_API\_tracing() {

enable\_api\_callback(&correlation\_dict);

enable\_activity(&records\_list);

}

void print\_collected\_trace() {

for (const auto& record : records\_list) {

correlation\_id\_t correlation\_id = record.correlation\_id;

std::string annotation = annotations\_dict[correlation\_id];

printf(“%s: [%lu, %lu]\n’, annotation.c\_str(), record.begin\_timestamp, record.end\_timestamp);

}

}

void main() {

enable\_API\_tracing();

<memcopy input data / submit kernels / memcopy output data>

print\_collected\_trace();

}