**ROC Profiler Library Specification**

rev 1.2.2

© Advanced Micro Devices Corp. 2017

All rights reserved.

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Authors** | **Description** |
| 1.0.0 | 8/31/2017 | Evgeny Shcherbakov | Initial version |
| 1.0.1 | 9/1/2017 | Evgeny Shcherbakov | Adding metric object type declaration and other changes |
| 1.1.0 | 9/18/2017 | Evgeny Shcherbakov | Renaming to rocprofiler library, context/info APIs, observer new/delete/simple APIs, code examples |
| 1.1.1 | 9/19/2017 | Evgeny Shcherbakov | Minor fixes of the code example 6.1 and others |
| 1.1.2 | 9/28/2017 | 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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Contents**

[**Revision history** 2](#_Toc497153964)

[**1.** **High level overview** 2](#_Toc497153965)

[**2.** **Library implementation requirements** 2](#_Toc497153966)

[**3.** **HSA runtime requirements** 3](#_Toc497153967)

[**4.** **Library basic usage flow** 4](#_Toc497153968)

[4.1. Profiling in the application context 4](#_Toc497153969)

[4.2. Statistical profiling / sampling, in the tool context, external to application context. 5](#_Toc497153970)

[4.3. Environment 5](#_Toc497153971)

[**5.** **Library API** 5](#_Toc497153972)

[5.1. Description 5](#_Toc497153973)

[5.2. Info API 6](#_Toc497153974)

[5.3. Context API 7](#_Toc497153975)

[5.4. Callback API 9](#_Toc497153976)

[5.5. Returning the error string method 9](#_Toc497153977)

[**6.** **Application code examples** 10](#_Toc497153978)

[6.1. Querying available metrics and trace parameters 10](#_Toc497153979)

[6.2. Profiling code example 10](#_Toc497153980)

[6.3. Option to use context callback 11](#_Toc497153981)

# **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 and thread traces (SQTT). The implementation distinguishes two profiling features, metrics and traces, where HW performance counters are treated as the basic metrics.

The library can be used by a tool library loaded by HSA runtime or by higher level HW independent performance analysis API like PAPI.

The library is written on C and will be based on AQLprofile AMD specific HSA extension. The library implementation requires HSA API intercepting and a profiling queue supporting a submit callback interface.

# **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)
  2. The library provides a C based PMC and SQTT profiling APIs to start, stop, read metrics results and tracing data.
  3. The library provides mechanism to load profiling tool library by env variable ROCP\_TOOL\_LIB
  4. The library provides methods to query the list of supported HW counters and the metrics name/id conversion methods.
  5. The library provides a callback API for enabling profiling and to read profiling data for the kernels dispatched to HSA AQL queues.
  6. The library is responsible for allocation of the buffers for profiling and notifying about output data buffer overflow for SQTT.
  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 and id/name conversion API
* Counters and metrics can be dynamically configured using XML configuration files with counters and metrics tables:
  + Counters table entry/basic metric: counter 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=L1.CYCLES/ L1.MISS ></metric>

</global>

# **HSA runtime requirements**

* 1. HSA runtime is supporting API intercepting mechanism so the ROC profiler library be able to intercept AQL queue create API and to return a profiling queue.
  2. Proxy queue supporting a kernel submit callback interface and will be used as a profiling queue implementation. The callback will be called for submitted valid AQL packets and will let to ROC profiler to return pre and post profiling packets.

# **Library basic usage flow**

## 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.

Profiling Tool

HSA application

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 API

HW counters start/stop/read

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

# **Library API**

## Description

The library provides the methods to start, stop and read HW performance counters PMC and thread traces SQTT. 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 a status code.

Returned API status:

* hsa\_status\_t – HSA status codes are used from hsa.h header

Info API:

* rocprofiler\_type\_t - profiling feature type
* rocprofiler\_parameter\_t – profiling feature parameter
* rocprofiler\_data\_kind\_t - profiling data kind
* rocprofiler\_data\_t - profiling data
* rocprofiler\_info\_t – profiling feature info
* rocprofiler\_get\_info – return querying info

Context API:

* rocprofiler\_mode\_t – profiling modes
* rocprofiler\_properties\_t – profilier properties
* rocprofiler\_t – profiling context
* rocprofiler\_open – create new profiling context
* rocprofiler\_close – delete profiling context and all allocated resources
* rocprofiler\_get\_groups – return profiling groups array
* rocprofiler\_start - start profiling
* rocprofiler\_stop - stop profiling
* rocprofiler\_sample - read profiling data to the info objects
* rocprofiler\_get\_data - method for iterating output data samples

Group versions of start/stop/sample/get\_data methods:

* + rocprofiler\_group\_start
  + rocprofiler\_group\_stop
  + rocprofiler\_group\_sample
  + rocprofiler\_get\_group\_data
* rocprofiler\_get\_metrics - method for iterating output data samples
* rocprofiler\_iterate\_trace\_data - method for iterating output trace data instances

Callback API:

* rocprofiler\_callback\_t – profiling callback type
* rocprofiler\_callback\_data\_t – profiling callback data type
* rocprofiler\_set\_dispatch\_observer – adding kernel dispatch observer
* rocprofiler\_remove\_dispatch\_observer – removing kernel dispatch observer

Returning the error string method:

* rocprofiler\_error\_string – method for returning the API error string

## Info API

Profiling metrics are defined by name and traces are defined by parameters. A metric unique id can be received by adding of the metric info to the profiling context or by ‘get\_info’ method. Where the name is a string and id is an integer uniquely identifying the counter. All supported metrics can be enumerated using ‘get\_info’ method. The counter names are defined in counters table configuration file, each counter is defined by unique name, block id and event id.

Profiling feature type:

typedef enum {

ROCPROFILER\_TYPE\_METRIC = 0,

ROCPROFILER\_TYPE\_TRACE = 1,

ROCPROFILER\_TYPE\_API = 2

} rocprofiler\_type\_t;

Profiling feature parameter:

typedef hsa\_ven\_amd\_aqlprofile\_parameter\_t rocprofiler\_parameter\_t;

Profiling data kind:

typedef enum {

ROCPROFILER\_UNINIT = 0,

ROCPROFILER\_INT32 = 1,

ROCPROFILER\_INT64 = 2,

ROCPROFILER\_FLOAT = 3,

ROCPROFILER\_DOUBLE = 4,

ROCPROFILER\_DATA = 5

} rocprofiler\_data\_kind\_t;

Profiling data type:

typedef struct {

rocprofiler\_metric\_kind\_t kind; // result kind

union {

uint32\_t result32; // 32bit integer result

uint64\_t result64; // 64bit integer result

float result\_float; // float single-precision result

double result\_double; // float double-precision result

typedef struct {

void\* ptr;

uint32\_t size;

uint32\_t instances; // number of trace instances

} result\_bytes; // data by ptr and byte size

};

} rocprofiler\_data\_t;

Profiling feature info:

typedef struct {

rocprofiler\_type\_t type; // feature type

const char\* name; // [in] feature name

uint32\_t id; // unique id;

const rocprofiler\_parameter\_t\* parameters; // [in] feature parameters

rocprofiler\_data\_t\* data; // [out] profiling data

} rocprofiler\_info\_t;

Profiling info type:

typedef enum {

ROCPROFILER\_INFO\_NEXT\_METRIC = 0,

ROCPROFILER\_INFO\_LAST\_METRIC = 1,

ROCPROFILER\_INFO\_NEXT\_PARAMETER = 2,

ROCPROFILER\_INFO\_LAST\_PARAMETER = 3,

ROCPROFILER\_INFO\_NEXT\_API = 4,

ROCPROFILER\_INFO\_LAST\_API = 5

} rocprofiler\_info\_t;

Get profiling info method:

has\_status\_t rocprofiler\_get\_info(

uint32\_t gpu\_index, // GPU device index

rocprofiler\_info\_t\* info); // [in/out] profiling info

## Context API

Profiling context is accumulating all profiling information including profiling features info, parameters and profiling 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. Using the context the profiling can be started and stopped in standalone mode or profiling can be initiated by registering dispatch callback.

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.

Profiling modes:

typedef struct {

ROCPROFILER\_MODE\_STANDALONE = 1, // Standalone mode when ROCProfiler supports own

// HSA AQL queue

ROCPROFILER\_MODE\_CREATEQUEUE = 2, // Profiler creates queue in STANDALONE mode

ROCPROFILER\_MODE\_SINGLEGROUP = 3 // Profiler allow one group only and fail 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:

typededf struct {

rocprofiler\_context\_callback\_t callback; // Callback on the context data readiness

void\* callback\_arg; // Callback arg

has\_queue\_t\* queue; // HSA queue

uint32\_t queue\_depth; // created queue depth

} rocprofiler\_properties\_t;

Open/close profiling context:

hsa\_status\_t rocprofiler\_open(

uint32\_t gpu\_index, // GPU device index

rocprofiler\_info\_t\* info, // [in/out] profiling info array

uint32\_t info\_count, // profiling info count

rocprofiler\_t\*\* context, // [out] context object

uint32\_t mode, // profiling mode mask

rocprofiler\_properties\_t\* properties); // profiler properties

hsa\_status\_t rocprofiler\_close(

rocprofiler\_t\* context); // [in] profiling context

Profiling group type:

typedef struct {

uint32\_t index; // profiling group index

rocprofiler\_info\_t\*\* info; // profiling info array

uint32\_t info\_count; // profiling info count

rocprofiler\_t\* context; // profiling context

} rocprofiler\_group\_t;

Return profiling groups array:

hsa\_status\_t rocprofiler\_get\_groups(

rocprofiler\_t\* context, // [in/out] profiling context

rocprofiler\_group\_t\*\* groups, // [out] groups array

uint32\_t\* group\_count); // [out] groups count

Start/stop/sample 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\_sample(

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/sample/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\_sample(

rocprofiler\_group\_t\* group); // [in/out] profiling group

hsa\_status\_t rocprofiler\_get\_group\_data(

rocprofiler\_group\_t\* group); // [in/out] profiling group

Read/calculate metrics data to the registered profiling info data fields:

hsa\_status\_t rocprofiler\_get\_metrics(

rocprofiler\_t\* context); // [in/out] profiling context

Method for iterating trace data instances:

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

## Callback API

The library provides a callback API for enabling profiling for the kernels dispatched to HSA AQL queues. Currently only kernel dispatch callback is supported which is a pseudo HSA API emulated by proxy queue.

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 {

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

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

uint32\_t gpu\_index; // GPU device index

} rocprofiler\_callback\_data\_t;

Adding/removing a kernel dispatch observer:

hsa\_status\_t rocprofiler\_add\_dispatch\_observer(

rocprofiler\_callback\_t callback, // dispatch callback

void\* data); // [in/out] callback data

hsa\_status\_t rocprofiler\_remove\_dispatch\_observer();

## Returning the error string method

hsa\_status\_t rocprofiler\_error\_string (

const char \*\* str); // [out] the API error string pointer returning

# **Application code examples**

## Querying available metrics and trace parameters

Printing all available metrics and trace parameters:

// Printing all available metrics

hsa\_status\_t status = HSA\_STATUS\_SUCCESS;

rocprofiler\_info\_t info{};

info.type = ROCPROFILER\_TYPE\_METRIC; // specify feature type as metric

while (status == HSA\_STATUS\_SUCCESS) {

status = rocprofiler\_get\_info(context, ROCPROFILER\_INFO\_NEXT\_METRIC, &info);

printf(“id %d, metric \”%s\”\n”, info->id, info->name);

}

// Printing all supported SQTT trace parameters

status = HSA\_STATUS\_SUCCESS;

info = {}

info.type = ROCPROFILER\_TYPE\_TRACE; // specify feature type as trace

info.name = “SQTT”; // specify the name of the feature

while (status == HSA\_STATUS\_SUCCESS) {

status = rocprofiler\_get\_info(context, ROCPROFILER\_INFO\_NEXT\_PARAMETER, &info);

printf(“id %d, parameter code %u\n”, info->id, info->parameters[0]);

}

## Profiling code example

Profiling of L1/L2 miss counters, average memory bandwidth and thread trace collecting:

hsa\_status\_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\_info\_t info\* = new rocprofiler\_info\_t[3];

// Tracing parameters

rocprofiler\_parameter\_t\* parameters = new rocprofiler\_parameter\_t[2];

// Setting profiling info

info[0].type = ROCPROFILER\_METRIC;

info[0].name = “L1\_MISS\_RATIO”;

info[1].type = ROCPROFILER\_METRIC;

info[1].name = “DRAM\_BANDWIDTH”;

info[2].type = ROCPROFILER\_TRACE;

info[2].parameters = parameters;

info[2].parameter\_count = 2;

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

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

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

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

// Creating profiling context

status = rocprofiler\_open(agent, info, 3, &context, ROCPROFILER\_MODE\_SINGLEGROUP, NULL);

<check status>

// Get profiling groups

rocprofiler\_group\_t\* groups;

uint32\_t group\_count;

status = rocprofiler\_get\_groups(context, &groups, &group\_count);

<check\_status>

// Profiling group 0

\*group = groups[0];

<saving of the profiling groups/profiling info>

return status;

}

void profiling\_libary\_constructor() {

hsa\_status\_t status;

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

void\* callback\_data = NULL;

// Adding observers

status = rocprofiler\_add\_dispatch\_observer(dispatch\_callback, callback\_data);

<check status>

// Dispatching profiled kernel

<dispatching profiled kernels>

}

void profiling\_libary\_destructor() {

<for entry : <saved profiling groups>> {

status = rocprofiler\_get\_group\_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->info, entry->info\_count);

}

}

## Option to use context callback

Creating profiling context with callback:

. . .

rocprofiler\_properties\_t properties = {};

properties.callback = dispatch\_callback;

properties.callback\_arg = NULL; // no args defined

status = rocprofiler\_open(agent, info, 3, &context, ROCPROFILER\_MODE\_SINGLEGROUP, properties);

<check status>

. . .

Definition of context callback:

void dispatch\_callback(profiler\_group\_t\* group, void\* arg) {

<tool\_dump\_data\_method>(group);

hsa\_status\_t status = rocprofiler\_close(group->context);

<check status>

}