**ROC Profiler Library Specification**

rev 1.2.3

© 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 |
| 1.2.3 | 11/22/2017 | Evgeny Shcherbakov | Renamings; gpu index changed on HSA agent; info iterate/query API |
|  |  |  |  |
|  |  |  |  |

**Contents**

[**Revision history** 2](#_Toc499137731)

[**1.** **High level overview** 2](#_Toc499137732)

[**2.** **Library implementation requirements** 3](#_Toc499137733)

[**3.** **HSA runtime requirements** 4](#_Toc499137734)

[**4.** **Library basic usage flow** 4](#_Toc499137735)

[4.1. Per kernel data collection, profiling in the application context 4](#_Toc499137736)

[4.2. Statistical profiling / sampling, in the tool context, external to application context. 5](#_Toc499137737)

[4.3. Environment 5](#_Toc499137738)

[**5.** **Library API** 5](#_Toc499137739)

[5.1. Description 5](#_Toc499137740)

[5.2. Info API 6](#_Toc499137741)

[5.3. Context API 7](#_Toc499137742)

[5.4. Sampling API 10](#_Toc499137743)

[5.5. Dispatch callback API 10](#_Toc499137744)

[5.6. Returning the error string method 11](#_Toc499137745)

[**6.** **Application code examples** 11](#_Toc499137746)

[6.1. Querying available metrics, traces, SQTT trace parameters 11](#_Toc499137747)

[6.2. Profiling code example 12](#_Toc499137748)

[6.3. Option to use completion callback 13](#_Toc499137749)

# **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 has C API and is based on AQLprofile AMD specific HSA extension. The library implementation requires HSA kernel dispatch intercepting support.

# **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 methods to query the list of supported HW features.
  3. The library provides PMC and SQTT 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 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 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
* 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.
  2. HSA runtime should provide mechanism for retrieving of mangled kernel name for a given kernel object address so the actual language dependent tool could mangle them to the actual kernel name.

# **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

# **Library 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.

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\_dispatch\_callback – adding kernel dispatch callback
* rocprofiler\_remove\_dispatch\_callback – removing kernel dispatch callback

Returning the error string method:

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

## 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 is defined by unique name, block name and event id.

Profiling info kind:

typedef enum {

ROCPROFILER\_INFO\_KIND\_METRIC = 0, // metric info

ROCPROFILER\_INFO\_KIND\_TRACE = 1, // trace info

ROCPROFILER\_INFO\_KIND\_TRACE\_PARAMETER = 2 // trace parameter info

} rocprofiler\_info\_kind\_t;

Profiling info query:

typedef union {

rocprofiler\_info\_kind\_t info\_kind; // queried profiling info kind

struct {

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

} trace\_parameter;

} rocprofiler\_info\_query\_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 praneter\_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;

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

has\_status\_t rocprofiler\_iterate\_info(

rocprofiler info\_kind\_t kind, // kind of iterated info

hsa\_status\_t (\**callback*)(const rocprofiler\_info\_data\_t info, void \*data), // callback

void \**data*); // data passed to callback

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

has\_status\_t rocprofiler\_query\_info(

rocprofiler\_info\_query\_t 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.

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 modes:

typedef struct {

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 = 3 // 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:

typededf 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 device

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 group:

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 fpr 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:

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

## 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

## Dispatch callback 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 a kernel dispatch callback:

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

rocprofiler\_callback\_t callback, // dispatch callback

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

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

## 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, 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);

x 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(

ROCPROFILER\_INFO\_KIND\_METRIC,

info\_data\_*callback,*

*NULL);*

*<check status>*

Printing all available traces:

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

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\_PARAMETER;

query.trace\_parameter.trace\_name = “SQTT”;

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

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\_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\_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(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>

}