roctracer Documentation

Release 4.1.0

Advanced Micro Devices, Inc.

Mar 27, 2026

INSTALL

Important

ROCTracer, ROCProfiler, `rocprof`, and `rocprofv2` are deprecated. It's strongly recommended to upgrade to the latest version of the ROCprofiler-SDK library and the rocprofv3 tool to ensure continued support and access to new features.

To learn about key feature improvements and benefits of ROCprofiler-SDK over the deprecated ROCProfiler and ROCTracer, see Comparing ROCprofiler-SDK to legacy ROCm profiling tools.

It's anticipated that ROCTracer, ROCProfiler, `rocprof`, and `rocprofv2` will reach end of support (EoS) by the end of 2026 Q2.

ROCTracer consists of the *ROCTracer* and *ROC-TX* libraries, which provide APIs to help you trace an application in the runtime. The runtime-independent API enables tracing runtime calls and asynchronous activities such as GPU kernel dispatches and memory moves to profile the application and hardware performance.

The code is open and hosted at https://github.com/ROCm/rocm-systems/tree/develop/projects/roctracer.

Note

The ROCTracer repository for ROCm 7.0 and earlier is located at https://github.com/ROCm/roctracer.

Install

- *Installing ROCTracer*

Reference

- *ROCTracer library specification*
- *ROC-TX library specification*
- *ROCTracer API library*
  - *Modules*
  - *Data Structures*
  - *Files*

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

You can find licensing information on the Licensing page.

CHAPTER

ONE

INSTALLING ROCTRACER

This topic provides information on installing ROCTracer.

1.1 Prerequisites

- Linux system supported by ROCm as described in system requirements
- ROCm software installed on the system
- Install packages as per your system:

Ubuntu

```
apt install python3 python3-pip gcc g++ libatomic1 make \
cmake doxygen graphviz texlive-full
```

RHEL

```
yum install -y python3 python3-pip gcc gcc-g++ make \
cmake libatomic doxygen graphviz texlive \
texlive-xtab texlive-multirow texlive-sectsty \
texlive-tocloft texlive-tabu texlive-adjustbox
```

SLES

```
zypper in python3 python3-pip gcc gcc-g++ make \
cmake libatomic doxygen graphviz \
texlive-scheme-medium texlive-hanging texlive-stackengine \
texlive-tocloft texlive-etoc texlive-tabu
```

- Install Python libraries such as *CppHeaderParser* and *argparse* for Python modules:

```
pip3 install CppHeaderParser argparse
```

1.2 Build and test from source

Follow these steps to build ROCTracer:

1. Clone development branch of `roctracer`:

```
git clone -b develop https://github.com/ROCm/rocm-systems.git
```

2. Build ROCTracer library:

```
cd <your path>/roctracer
./build.sh
```

3. Build and run test:

```
cd <your path>/roctracer/build
make mytest
run.sh
```

1.3 Installation

Install using:

```
make install
```

or

```
make package && dpkg -i *.deb
```

1.4 Sample programs

- MatrixTranspose-test.cpp
- MatrixTranspose.cpp

CHAPTER

TWO

ROCTRACER LIBRARY SPECIFICATION

The ROCTracer library provides runtime-independent APIs for tracing runtime calls and asynchronous activities such as GPU kernel dispatches and memory moves. The tracing includes callback APIs for runtime API tracing and activity APIs for asynchronous activity records logging. Use these APIs to develop tracing tools or to implement tracing in applications.

The ROCTracer library provides runtime-independent APIs for tracing the runtime calls and asynchronous activities such as GPU kernel dispatches and memory moves. The tracing includes callback APIs for runtime API tracing and activity APIs for asynchronous activity records logging. The activity tracing results are recorded in a ring buffer.

Depending on the runtime intercepting mechanism, the ROCTracer library can be dynamically linked and loaded by the runtime as a plugin, or an API wrapper can be loaded using *LD_PRELOAD*. The library has a C API.

Note

The ROCTracer library provides rocTracer API version 2.

2.1 Using the rocTracer API

To use the rocTracer API, link the application with ROCTracer using the API header and dynamic library:

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

2.2 General APIs

The general APIs enlist the methods used to get the error code and error string of the last failed library API call. This helps to check for the successful completion of the library API call.

2.2.1 Error codes and error strings

The error codes are defined in the enumeration as:

```
typedef enum {
   ROCTRACER_STATUS_SUCCESS = 0,
   ROCTRACER_STATUS_ERROR = 1,
   ROCTRACER_STATUS_UNINIT = 2,
   ROCTRACER_STATUS_BREAK = 3,
   ROCTRACER_STATUS_BAD_DOMAIN = 4,
   ROCTRACER_STATUS_BAD_PARAMETER = 5,
```

(continues on next page)

(continued from previous page)

```
   ROCTRACER_STATUS_HIP_API_ERR = 6,
   ROCTRACER_STATUS_HCC_OPS_ERR = 7,
   ROCTRACER_STATUS_ROCTX_ERR = 8,
} roctracer_status_t;
```

Method to get the error string:

```
const char* roctracer_error_string();
```

2.2.2 Library version

The ROCTracer library provides the major version for incompatible API changes and the minor version for bug fixes.

API version macros are defined in the library API header 'roctracer.h' as:

```
ROCTRACER_VERSION_MAJOR
ROCTRACER_VERSION_MINOR
```

Methods to check library major and minor versions:

```
uint32_t roctracer_major_version();
uint32_t roctracer_minor_version();
```

2.3 Frontend API

ROCTracer provides support for runtime API callbacks and activity records logging. The APIs of different runtimes at different levels such as the language level and the driver level are considered as different API domains with assigned domain IDs. The API callbacks provide the arguments for the API call and are called on "enter" and "exit" stages. The activity records are logged to the ring buffer and can be associated with the respective API calls using the correlation ID. The activity APIs are used to enable collection of records with timestamping data for API calls and asynchronous activities such as kernel submits, memory copies and barriers.

2.3.1 Tracing domains

ROCTracer provides APIs to trace various domains such as HSA, HIP, and HCC runtime levels with each domain assigned with a domain ID. The domains and their IDs are defined in the enumeration as shown below:

```
typedef enum {
   ACTIVITY_DOMAIN_HSA_API = 0,         // HSA API domain
   ACTIVITY_DOMAIN_HSA_OPS = 1,         // HSA async activity domain
   ACTIVITY_DOMAIN_HIP_API = 2,         // HIP API domain
   ACTIVITY_DOMAIN_HIP_OPS = 3,         // HIP async activity domain
   ACTIVITY_DOMAIN_KFD_API = 4,         // AMD Kernel-mode GPU Driver (KMD) API domain
   ACTIVITY_DOMAIN_EXT_API = 5,         // External ID domain
   ACTIVITY_DOMAIN_ROCTX   = 6,         // ROCTX domain
   ACTIVITY_DOMAIN_NUMBER = 7
} activity_domain_t;
```

Method to return Op string for the given domain and activity Op code:

```
const char* roctracer_op_string(  // NULL returned on error and error number is set
   uint32_t domain,            // tracing domain
```

(continues on next page)

(continued from previous page)

```
   uint32_t op,              // activity op code
   uint32_t kind);        // activity kind
```

Method to retun Op code and kind for the given Op string:

```
roctracer_status_t roctracer_op_code(
   uint32_t domain,              // tracing domain
   const char* str,              // [in] op string
   uint32_t* op,                 // [out] op code
   uint32_t* kind);              // [out] op kind code if not NULL
```

2.3.2 Callback APIs

ROCTracer provides support for runtime API callbacks and activity records logging. The API callbacks provide API call arguments and are called during "enter" and "exit" phases.

The enumeration defining the API phase to be passed to the callbacks:

```
typedef enum {
   ROCTRACER_API_PHASE_ENTER,
   ROCTRACER_API_PHASE_EXIT,
} roctracer_api_phase_t;
```

Runtime API callback type:

```
typedef void  (*roctracer_rtapi_callback_t)(
   uint32_t domain,   // runtime API domain
   uint32_t cid,      // API call ID
   const void* data,  // [in] callback data with correlation ID and the call arguments
   void* arg);        // [in/out] value to be passed by the user
```

Method to enable runtime API callbacks for the given domain and Op code:

```
roctracer_status_t roctracer_enable_op_callback(
   activity_domain_t domain,             // tracing domain
   uint32_t op,                          // API call ID
   activity_rtapi_callback_t callback,   // callback function pointer
   void* arg);                           // [in/out] value to be passed by the user
```

Method to enable runtime API callback for all Ops in the given domain:

```
roctracer_status_t roctracer_enable_domain_callback(
   activity_domain_t domain,             // tracing domain
   activity_rtapi_callback_t callback,   // callback function pointer
   void* arg);                          // [in/out] value to be passed by the user
```

Method to enable runtime API callback for all domains and all Ops:

```
roctracer_status_t roctracer_enable_callback(
   activity_rtapi_callback_t callback,   // callback function pointer
   void* arg);                           // [in/out] value to be passed by the user
```

Method to disable runtime API callback for the given domain and Op code:

```
roctracer_status_t roctracer_disable_op_callback(
   activity_domain_t domain,           // tracing domain
   uint32_t op);                       // API call ID
```

Method to disable runtime API callback for all Ops in the given domain:

```
roctracer_status_t roctracer_disable_domain_callback(
   activity_domain_t domain);          // tracing domain
```

Method to disable runtime API callback for all domains and all Ops:

```
roctracer_status_t roctracer_disable_callback();
```

2.3.3 Activity APIs

The activity records are asynchronously logged to the pool and can be associated with the respective API callbacks using the correlation ID. You can use the activity APIs to enable collection of records with timestamp data for API calls and GPU activities such as kernel submits, memory copies, and barriers.

Correlation ID type:

```
typedef uint64_t activity_correlation_id_t;
```

Activity record type:

```
struct activity_record_t {
   uint32_t domain;                           // activity domain ID
   activity_kind_t kind;                      // activity kind
   activity_op_t op;                          // activity op
   activity_correlation_id_t correlation_id;  // activity ID
   uint64_t begin_ns;                         // host begin timestamp
   uint64_t end_ns;                           // host end timestamp
   union {
      struct {
         int device_id;                       // device ID
         uint64_t queue_id;                   // queue ID
      };
      struct {
         uint32_t process_id;                 // device ID
         uint32_t thread_id;                  // thread ID
      };
      struct {
      activity_correlation_id_t external_id; // external correlation ID
      };
   };
   size_t bytes;                              // data size bytes
};
```

Method to return next record:

```
static inline int roctracer_next_record(
   const activity_record_t* record,         // [in] record pointer
   const activity_record_t** next);         // [out] next record pointer
```

ROCTracer allocator type:

```
typedef void (*roctracer_allocator_t)(
   char** ptr,               // memory pointer
   size_t size,        // memory size
   void* arg);         // allocator argument
```

Pool callback type:

```
typedef void (*roctracer_buffer_callback_t)(
   const char* begin,   // [in] beginning of buffered trace records
   const char* end,     // [in] end of buffered trace records
   void* arg);          // [in/out] value to be passed by the user
```

ROCTracer properties:

```
typedef struct {

/**
   * ROCTracer mode
   */
uint32_t mode;

/**
   * Size of buffer in bytes
   */
size_t buffer_size;

/**
   * The allocator function for allocation and deallocation of the buffer. If NULL then␣
↪malloc, realloc, and free are used.
   */
roctracer_allocator_t alloc_fun;

/**
   * The argument required to invoke the alloc_fun allocator
   */
void* alloc_arg;

/**
   * The function that needs to be called when a buffer becomes full or is flushed.
   */
roctracer_buffer_callback_t buffer_callback_fun;

/**
   * The argument required to invoke the buffer_callback_fun callback.
   */
void* buffer_callback_arg;

} roctracer_properties_t;
```

ROCTracer memory pool handle type:

```
typedef void roctracer_pool_t;
```

Methods to create ROCTracer memory pool:

This method sets the created memory pool along with the specified properties as the default memory pool.

```
roctracer_status_t roctracer_open_pool(
   const roctracer_properties_t* properties); // ROCTracer pool properties
```

This method returns handle to the newly created memory pool. If pool is not allocated then it sets the newly created pool with the specified properties as the default pool.

```
roctracer_status_t roctracer_open_pool_expl(
   const roctracer_properties_t* properties, // ROCTracer pool properties
   roctracer_pool_t** pool);                 // [out] returns tracer pool if not NULL␣
↪otherwise sets the
                                             // default one if it is not set else␣
↪generates an error
```

Methods to close ROCTracer memory pool:

Before closing the pool, ensure that all enabled activities that use the pool have finished writing to the pool. Closing a pool automatically disables any activities that specify the pool and flushes it.

This method closes the default memory pool if defined and sets it to undefined.

```
roctracer_status_t roctracer_close_pool();
```

This method allows you to specify the memory pool to be closed. If no pool is specified, then the close operation is performed on the default pool.

```
roctracer_status_t roctracer_close_pool_expl(
   roctracer_pool_t* pool);          // memory pool. A NULL value means default pool
```

Methods to return current default pool:

This method queries the current default memory pool.

```
roctracer_pool_t* roctracer_default_pool();
```

This method queries and sets new default pool if the argument is not NULL.

```
roctracer_pool_t* roctracer_default_pool_expl(
   roctracer_pool_t* pool);          // new default pool if not NULL
```

Activity records logging

You can enable activity records logging for a specific operation of a domain that utilizes a memory pool.

Methods to enable activity records logging:

This method enables activity records logging using the default pool for the given operation of the given domain.

```
roctracer_status_t roctracer_enable_op_activity(
   activity_domain_t domain,         // tracing domain
   uint32_t op);                     // activity op ID
```

This method enables activity records logging for the given operation of the given domain utilizing the given memory pool.

```
roctracer_status_t roctracer_enable_op_activity_expl(
   activity_domain_t domain,         // tracing domain
   uint32_t op,                      // activity op ID
   roctracer_pool_t* pool);          // memory pool where a NULL value points to the␣
↪default pool
```

This method enables activity records logging using the default pool for all the operations of the given domain.

```
roctracer_status_t roctracer_enable_domain_activity(
   activity_domain_t domain);        // tracing domain
```

This method enables activity records logging for all the operations of the given domain utilizing the given memory pool.

```
roctracer_status_t roctracer_enable_domain_activity_expl(
   activity_domain_t domain,         // tracing domain
   roctracer_pool_t* pool);          // memory pool where a NULL value points to the␣
↪default pool
```

This method enables acitivity records logging using the default pool for all the operations of all the domains.

```
roctracer_status_t roctracer_enable_activity();
```

This method enables acitivity records logging for all the operations of all the domains utilizing the given memory pool.

```
roctracer_status_t roctracer_enable_activity_expl(
   roctracer_pool_t* pool);          // memory pool where a NULL value points to the␣
↪default pool
```

Methods to disable activity records logging:

This method disables activity records logging for the given operation of the given domain.

```
roctracer_status_t roctracer_disable_op_activity(
   activity_domain_t domain,         // tracing domain
   uint32_t op);                     // activity op ID
```

This method disables acitivity records logging for all the operations of the given domain.

```
roctracer_status_t roctracer_disable_domain_activity(
   activity_domain_t domain);        // tracing domain
```

Methods to flush available activity records:

If an activity record is still being written, flushing stops. To resume the flush, use a subsequent flush when the operation to write the record is complete.

This method flushes available activity records for the default memory pool.

```
roctracer_status_t roctracer_flush_activity();
```

This method flushes available activity records for the specified memory pool.

```
roctracer_status_t roctracer_flush_activity_expl(
   roctracer_pool_t* pool);          // memory pool. NULL points to the default pool
```

Method to return correlated GPU/CPU system timestamp:

```
roctracer_status_t roctracer_get_timestamp(
   uint64_t* timestamp);           // [out] return timestamp
```

External API association

These APIs provide activity records to associate ROCTracer correlation IDs with IDs provided by external APIs. The external ID records are identified by *ACTIVITY_DOMAIN_EXT_API* domain value. An external ID record is inserted before any generated ROCTracer activity record if the same CPU external ID stack is non-empty.

Method to push an external ID to a per CPU thread stack: This method notifies that the calling thread is entering an external API region.

```
roctracer_status_t roctracer_activity_push_external_correlation_id(
   activity_correlation_id_t id);        // external correlation Id
```

Method to pop the last pushed external ID from the CPU thread stack: This method notifies that the calling thread is leaving an external API region.

```
roctracer_status_t roctracer_activity_pop_external_correlation_id(
   activity_correlation_id_t* last_id);  // returns the last external correlation ID if␣
↪not NULL
```

Tracing control

The following APIs allow you to start or stop tracing.

Method to start tracing:

```
void roctracer_start();
```

Method to stop tracing:

```
void roctracer_stop();
```

2.4 Sample codes

This sample code demonstrates the HIP API, HCC ops, and GPU activity tracing.

```
#include <roctracer_hip.h>

// HIP API callback function
void hip_api_callback(
   uint32_t domain,
   uint32_t cid,
   const void* callback_data,
   void* arg)
{
   (void)arg;
   const hip_api_data_t* data = reinterpret_cast <const hip_api_data_t*>
   (callback_data);
   fprintf(stdout, "<%s id(%u)\tcorrelation_id(%lu) %s> ",
         roctracer_id_string(ACTIVITY_DOMAIN_HIP_API, cid),
         cid,
```

(continues on next page)

(continued from previous page)

```
        data->correlation_id,
      (data->phase == ACTIVITY_API_PHASE_ENTER) ? "on-enter" : "on-exit");
   <some code  . . .>
}

// Activity tracing callback
void activity_callback(const char* begin, const char* end, void* arg) {
   const roctracer_record_t* record = reinterpret_cast<const
                                    roctracer_record_t*>(begin);
   const roctracer_record_t* end_record = reinterpret_cast<const
                                      roctracer_record_t*>(end);
   fprintf(stdout, "\tActivity records:\n");
   while (record < end_record) {
      const char * name = roctracer_op_string(record->domain,
                                           record->activity_id, 0);
      fprintf(stdout, "\t%s\tcorrelation_id(%lu) time_ns(%lu:%lu)
            device_id(%d) stream_id(%lu)\n",
            name,
            record->correlation_id,
            record->begin_ns,
            record->end_ns,
            record->device_id,
            record->stream_id
            );
      <some code . . .>
      ROCTRACER_CALL(roctracer_next_record(record, &record));
   }
}

int main() {
   // Allocating tracing pool
   roctracer_properties_t properties{};
   properties.buffer_size = 12;
   properties.buffer_callback_fun = activity_callback;
   ROCTRACER_CALL(roctracer_open_pool(&properties));

   // Enable HIP API callbacks. HIP_API_ID_ANY can be used to trace all HIP API calls.
   ROCTRACER_CALL(roctracer_enable_op_callback(ACTIVITY_DOMAIN_HIP_API,
                                    HIP_API_ID_hipModuleLaunchKernel,
                                    hip_api_callback, NULL));
   ROCTRACER_CALL(roctracer_enable_op_activity(ACTIVITY_DOMAIN_HIP_API,
                                    HIP_API_ID_hipModuleLaunchKernel));
   // Enable HIP kernel dispatch activity tracing
   ROCTRACER_CALL(roctracer_enable_op_activity(ACTIVITY_DOMAIN_HIP_OPS,
                                          HIP_OP_ID_DISPATCH));

   <test code>

   // Disable tracing and close the pool
   ROCTRACER_CALL(roctracer_disable_callback());
   ROCTRACER_CALL(roctracer_disable_activity());
   ROCTRACER_CALL(roctracer_close_pool());
```

(continues on next page)

(continued from previous page)

```
}
```

Note

To demonstrate the use of the ROCTracer API, this document refers to the MatrixTranspose application as an example. A version of this application using the ROCTracer API is available in the roctracer/test/app folder on GitHub.

CHAPTER

THREE

ROC-TX LIBRARY SPECIFICATION

In certain situations, such as debugging performance issues in large-scale GPU programs, API-level tracing might be too fine-grained to provide a big picture of the program execution. In such cases, defining specific tasks to be traced is helpful.

To specify the tasks for tracing, enclose the respective source code with API calls from the ROC-TX library. This process is also known as instrumentation. As the scope of code for instrumentation is defined using the enclosing API calls, it is named a range. A range is a programmer-defined task with a well-defined start and end code scope. You can also fine-grain the scope specified within a range using further nested ranges.

Note

The ROC-TX library provides rocTX API version 1.

3.1 Using the rocTX API

To use the rocTX API, link the application with ROC-TX using the API header and dynamic library:

- **API header:** `/opt/rocm-{version}/roctracer/include/roctx.h`
- **Dynamic library (.so):** `/opt/rocm-{version}/lib/libroctx64.so.<version major>`

3.2 High-level overview

ROC-TX library contains application code instrumentation APIs to support high-level correlation of runtime API or activity events.

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.

```
roctxMark("before hipLaunchKernel");
```

- *roctxRangeStart*: Starts a range. Different threads can start ranges.

```
roctx_range_id_t roctx_id = roctxRangeStartA("roctx_range with ID");
```

- *roctxRangePush*: Starts a new nested range.

```
roctxRangePush("ROCTX-RANGE: hipLaunchKernel");
```

- *roctxRangePop*: Stops the current nested range.

```
roctxRangePop();
```

- *roctxRangeStop*: Stops the given range.

```
roctxRangeStop(roctx_id);
```

3.3 Sample code

To demonstrate the use of rocTX API with various options, this document refers to the MatrixTranspose application as an example.

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

CHAPTER

FOUR

ROCTRACER API

4.1 Related Pages

4.2 Modules

4.3 Data Structures

4.3.1 Data Structures

4.3.2 Data Structure Index

4.3.3 Data Fields

Data Fields

Data Fields - Variables

4.4 Files

4.4.1 File List

4.4.2 Globals

Globals

Globals

Globals

Globals

Globals

Globals

CHAPTER

FIVE

LICENSE