

figure

Contents

1	Overview	1
2	Counters and Metric Collection	3
3	Application Tracing	6
3.1	HSA Trace	7
3.2	HIP Trace	8
3.3	Sys Trace	9
3.4	Roctx Trace	9
3.5	Tracing Control	11
3.5.1	Filtering by event names	11
3.5.2	Code Annotation	11
3.5.3	Flush rate	12
4	Known Issues	12
5	Errors	12
6	Planned features	12

List of Figures

1	HSA Trace Visualization	7
2	HIP Trace Visualization	9
3	Sys Trace Visualization.	9
4	Roctx Trace Visualization.	11

1 Overview

rocprof is a command line tool that is implemented on the ROCm's profiling libraries, rocProfiler and rocTracer.

The input to this tool is a XML or a text file which contains counters list or trace parameters. Its output would be profiling data and statistics in various formats as text, CSV and JSON traces.

There are two modes in which this tool can be used : Counters and Metrics collection and Application Tracing. In the further sections, we would be discussing these two operating modes of the tool in detail. We would also be using the below MatrixTranspose example [1] to understand the usage of rocprof tool.

```

#include <iostream>

// hip header file
#include <hip/hip_runtime.h>
#include "roctracer_ext.h"
// roctx header file
#include <roctx.h>

#define WIDTH 1024
#define NUM (WIDTH * WIDTH)
#define THREADS_PER_BLOCK_X 4
#define THREADS_PER_BLOCK_Y 4
#define THREADS_PER_BLOCK_Z 1
// Mark API
extern "C"

// Device (Kernel) function, it must be void
__global__ void matrixTranspose(float* out, float* in, const int width)
{
    int x = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    int y = hipBlockDim_y * hipBlockIdx_y + hipThreadIdx_y;
    out[y * width + x] = in[x * width + y];
}

int main() {
    float* Matrix;
    float* TransposeMatrix;
    float* cpuTransposeMatrix;
    float* gpuMatrix;
    float* gpuTransposeMatrix;
    hipDeviceProp_t devProp;
    hipGetDeviceProperties(&devProp, 0);
    std::cout << "Device name " << devProp.name << std::endl;
    int i;
    int errors;
    Matrix = (float*)malloc(NUM * sizeof(float));
    TransposeMatrix = (float*)malloc(NUM * sizeof(float));
    // initialize the input data
    for (i = 0; i < NUM; i++) {
        Matrix[i] = (float)i * 10.0f;
    }
    // allocate the memory on the device side
    hipMalloc((void**)&gpuMatrix, NUM * sizeof(float));
    hipMalloc((void**)&gpuTransposeMatrix, NUM * sizeof(float));
    uint32_t iterations = 100;
    while (iterations-- > 0) {
        std::cout << "## Iteration (" << iterations << ")
        ##### << std::endl;
        // Memory transfer from host to device
        hipMemcpy(gpuMatrix, Matrix, NUM * sizeof(float),
        hipMemcpyHostToDevice);

        roctxMark("before hipLaunchKernel");
        int rangeId = roctxRangeStart("hipLaunchKernel range");
        roctxRangePush("hipLaunchKernel");
        // Launching kernel from host

```

```

hipLaunchKernelGGL(matrixTranspose, dim3(WIDTH /
    THREADS_PER_BLOCK_X,
    WIDTH / THREADS_PER_BLOCK_Y),
    dim3(THREADS_PER_BLOCK_X,
    THREADS_PER_BLOCK_Y), 0, 0,
    gpuTransposeMatrix,
    gpuMatrix, WIDTH);

roctxMark("after hipLaunchKernel");
// Memory transfer from device to host
roctxRangePush("hipMemcpy");
hipMemcpy(TransposeMatrix, gpuTransposeMatrix, NUM *
    sizeof(float), hipMemcpyDeviceToHost);
roctxRangePop(); // for "hipMemcpy"
roctxRangePop(); // for "hipLaunchKernel"
roctxRangeStop(rangeId);
}
// free the resources on device side
hipFree(gpuMatrix);
hipFree(gpuTransposeMatrix);
// free the resources on host side
free(Matrix);
free(TransposeMatrix);
return errors;
}

```

2 Counters and Metric Collection

This mode is used to collect hardware counters and metrics. Two types of counters that are available for the profiling are : basic and derived. The hardware counters are called the basic counters and the list of supported basic counters can be obtained from the following command:

```
rocprof -list-basic
```

The derived metrics are defined on top of the basic counters using mathematical expression. The supported derived metrics along with their mathematical expressions can be listed using the following command:

```
rocprof -list-derived
```

Additionally, profiling scope can be defined in the input file through GPU id list, kernel name substrings list, and dispatch range. Counters and metrics can be dynamically configured using XML configuration files with counters and metrics tables. The basic metric entry in the counters table looks like the following, basic metric: counter name, block name, event id. The derived metric entry has metric name and an expression for its calculation in the derived metrics table.

Below is a listing showing sample metrics both basic (gfx_metrics.xml) and derived (metrics.xml) XML files. The default metrics files can be found at rocprofiler/test/tool.

- gfx_metrics.xml

```

<gfx90a>
.....
<metric name="TCC_MISS" block=TCC event=19
descr="Number of cache misses.
UC reads count as misses."></metric>
.....
</gfx90a>

```

- metrics.xml

```

<gfx90a_expr>
...
</gfx90a_expr>
<global>
.....
<metric
name="MemUnitStalled"
descr="The percentage of GPU time the memory
unit is stalled.
Try reducing the number or size of fetches
and writes if possible.
Value range: 0% (optimal) to 100% (bad)."
expr=100*max(TCP_TCP_TA_DATA_STALL_CYCLES,16)/
GRBM_GUI_ACTIVE/SE_NUM>
</metric>
.....
</global>

```

Below is a sample input file for counter and metric collection mode.

```

# Perf counters group 1
pmc : MemUnitStalled,TCC_MISS[0]

# Filter by dispatches range, GPU index and kernel names
# supported range formats: "3:9", "3:", "3"
range: 0 : 1
gpu: 0
kernel: matrixTranspose

```

The rows in the text file starting with "pmc" are a group of metrics that the user is interested in collecting. The number of metrics that can be collected in one run of profiling is limited by GPU hardware. Hence, to collect metrics exceeding hardware limits, multiple pmc rows should be used to input different group of metrics. Collection of different metric groups can be achieved with application replays.

rocpf tool can provide suggestions to group the metrics incase the user has it exceeding the hardware limits. The user can use this suggestion to split the metrics into group sets and successfully perform metric collection. Below listing shows one such example where rocpf is proposing metric group set when user has an input with metrics exceeding hardware limit.

```

$cat input.txt
# Perf counters group 1

```

```

pmc : Wavefronts, VALUInsts, SALUInsts, SFetchInsts,
      FlatVMemInsts,
      LDSInsts, FlatLDSInsts, GDSInsts, VALUUtilization, FetchSize,
      WriteSize, L2CacheHit, VWriteInsts, GPUBusy, VALUBusy, SALUBusy,
      MemUnitStalled, WriteUnitStalled, LDSBankConflict, MemUnitBusy
# Filter by dispatches range, GPU index and kernel names
# supported range formats: "3:9", "3:", "3"
range: 0 : 1
gpu: 0
kernel:matrixTranspose

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

```

GPU blocks are implemented as several identical instances. To dump counters of specific instance, square brackets can be used as shown in the input sample.

The row starting with range identifies the range of kernel dispatches. In this example, we only have one kernel to profile hence the range would be 0:1.

The GPU(s) on which the hardware counters are collected on is identified by the row starting with "gpu". This enables the support for profiling multiple GPUs.

The row starting with kernel identifies the name(s) of the kernel needed to be profiled.

An output CSV is generated with the counter information.

rocpfprof usage example: User can profile the MatrixTranspose 1 application with the sample input file like follows:

```

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

```

The following is the description of each field in this output file.

- Index - kernels dispatch order index
- KernelName - the dispatched kernel name
- gpu-id - GPU id the kernel was submitted to
- queue-id - the ROCm queue unique id the kernel was submitted to
- queue-index - The ROCm queue write index for the submitted AQL packet
- tid - system application thread id which submitted the kernel

- grd - the kernel's grid size
- wgr - the kernel's work group size
- lds - the kernel's LDS memory size
- scr - the kernel's scratch memory size
- vgpr - the kernel's VGPR size
- sgpr - the kernel's SGPR size
- fbar - the kernel's barriers limitation
- sig - the kernel's completion signal

User can also get the timestamp columns such as DispatchNs/BeginNs/EndNs/-CompleteNs by enabling the timestamp option as follows:

```
$ rocprof -i input.txt --timestamp on ./MatrixTranspose
$ cat input.csv
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,
wgr,lds,scr,vgpr,sgpr,fbar,sig,obj,
MemUnitStalled,TCC_MISS[0],DispatchNs,BeginNs,EndNs,CompleteNs
0,"matrixTranspose(float*, float*, int) [clone .kd]",0,0,0,2837,2837,
1048576,16,0,0,8,24,0,0x0,
0x7fcd75984580,5.9792124305,4096,87851328156768,
87851334047658,87851334141098,87851334732528
```

User can provide rocprof with a preferred name for the output CSV file as follows.

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

3 Application Tracing

Application tracing is used to collect runtime (ROCr and HIP) API callbacks and activities: kernel execution, async memory copy, barrier packets. The trace consists of several sections with timelines for API trace per thread and activity. The following are the types of traces supported by the tool.

3.1 HSA Trace

HSA trace includes ROCr API callbacks and activities tracing at AQL queue level. The trace file would have entries for each HSA call. The GPU activity is collected into a separate text file. User can generate the HSA trace of MatrixTranspose 1 application with the following command.

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

The above command generates the following output files:

- results.csv - The file is same as the one discussed in Listing 2
- results.json : A JSON file which can be used to visualize the trace information. A third party tool such as Perfetto [2] can be used to visualize the JSON file. Figure 1 is using Perfetto [2]. Here sections, HSA API



Figure 1: HSA Trace Visualization

for APIs and COPY for GPU activity, async_ memory_ copy can be seen. The section, GPU ID is for the kernels executed on that GPU.

- Statistics files: results.stats.csv, results.hsa_stats.csv, results.copy_stats.csv. These are the statistics file that have kernel, HSA API and activity statistics respectively. The fields in these files are :
 - Name
 - Calls
 - TotalDurationNs
 - AverageNs
 - Percentage

All the profiling data is consolidated into results.json file. Intermediary profiling data is deleted. However, user can save this data by providing a directory using -d option.

```
$rocprof -d outputFolder --hsa-trace ./MatrixTranspose
```

The output folder will mainly have the following files

- hsa_ api_ trace.txt: This file contains the HSA API trace data. The fields in this file are :

- API timelines
- host process identifier
- host thread identifier
- API name, arguments, and its return code
- correlation-id

Here is an entry from hsa_ api_ trace.txt file of the MatrixTranspose 1 application.

```
$cat outputFolder/rpl../input../hsa_api_trace.txt
155094092079246:155094092081176 10842:10842
hsa_system_get_major_extension_table(513, 1, 24, 0
    x7f5f22ce80c0) =
0 :10
.....
.....
```

- async_ copy _ trace.txt : This file contains the GPU Activity i.e. async_ memory_ copy trace information. The fields in this file are :

- Activity timeline
- Activity name
- correlation id
- host process identifier

Here is an entry from async_ copy_ trace.txt of the MatrixTranspose application.

```
$cat outputFolder/rpl../input../async_copy_trace.txt
155094395212562:155094395435602 async-copy:441:10842
.....
.....
```

3.2 HIP Trace

HIP trace includes HIP API timelines and HIP activity at the runtime level. HIP trace of Matrixtranspose 1 application can be generated using the below command.

```
rocprof usage: rocprof -d outputFolder --hip-trace ./Matrixtranspose
$cat outputFolder/rpl../input../hip_api_trace.txt
151604814578343:151604814635562 10774:10774
hipMalloc(ptr=0x7f0a5b400000, size=4194304) :2
....
....
$cat outputFolder/rpl../input../hcc_ops_trace.txt
151605101732782:151605101957902 0:0 CopyHostToDevice:4:10774
.....
.....
```

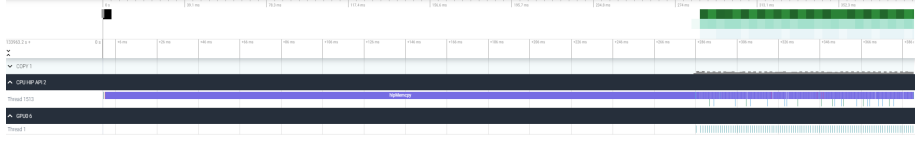


Figure 2: HIP Trace Visualization

This generates the results.json which can be visualized. Figure 2 is HIP trace visualization using Perfetto [2]. In the Figure 2 we see sections, HIP (API for APIs) and COPY (for copy operations between host and device). The HIP API trace is written into hip_api_trace.txt. The HIP activity trace is written into hcc_ops_trace.txt. The fields in API and activity files are similar to those in HSA. An entry from each of these files is shown in Listing 3.2.

Along with the trace files the following files are generated.

- results.json - JSON file that can be used to visualize the trace.
- statistics files: results.stats.csv, results.hip_stats.csv, results.copy_stats.csv. These files have kernel statistics, HIP API and activity statistics respectively.

3.3 Sys Trace

Generates both HSA and HIP trace together. The following is the command line option to generate sys trace file.

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

Figure 3 is the sys trace visualization using Perfetto [2]. We can see sections from both HIP and HSA trace here.

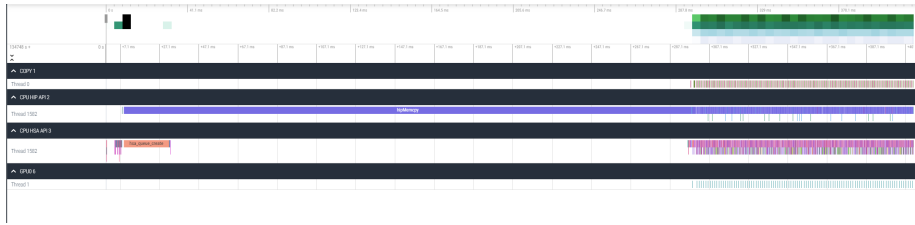


Figure 3: Sys Trace Visualization.

3.4 Roctx Trace

Generates traces for code annotations. The roctx trace information is generated in a file, roctx_trace.txt in the output folder. The file includes

entries for every marker and range that is inserted into the code. The file has fields such as timelines, process identifier, thread identifier, range identifier, and the message. The marker and ranges in this application is inserted with the following API.

Below, we highlight the section of Listing 1 that has roctx markers and ranges.

```
roctxMark("before hipLaunchKernel");
int rangeId = roctxRangeStart("hipLaunchKernel range");
roctxRangePush("hipLaunchKernel");
// Launching kernel from host
hipLaunchKernelGGL(matrixTranspose, dim3(WIDTH /
    THREADS_PER_BLOCK_X,
    WIDTH / THREADS_PER_BLOCK_Y),
    dim3(THREADS_PER_BLOCK_X,
    THREADS_PER_BLOCK_Y), 0, 0,
    gpuTransposeMatrix,
    gpuMatrix, WIDTH);

roctxMark("after hipLaunchKernel");
// Memory transfer from device to host
roctxRangePush("hipMemcpy");
hipMemcpy(TransposeMatrix, gpuTransposeMatrix, NUM *
    sizeof(float), hipMemcpyDeviceToHost);
roctxRangePop(); // for "hipMemcpy"
roctxRangePop(); // for "hipLaunchKernel"
roctxRangeStop(rangeId);
```

- roctxMark : Inserts a marker in the code with a message.
- roctxRangeStart : Starts a range. Ranges can be started by different threads.
- roctxRangePush : Starts a new nested range.
- roctxRangePop : Stops the current nested range.
- roctxRangeStop : Stops the given range.

Below, is the command to obtain roctx trace.

```
$Rocprof usage: rocprof -d outputFolder --roctx-trace ./
    MatrixTranspose
$cat outputFolder/rpl.../input../roctx_trace.txt
139496837144269 4811:4811 0:0:"before hipLaunchKernel"
....
....
```

In Figure 4, we show the result of visualizing the roctx trace on Perfetto tool [2] using the generated results.json file. The section, Markers and Ranges show the marked events and ranges. User can also control the tracing. Next, we would be discussing the ways to control tracing in detail.

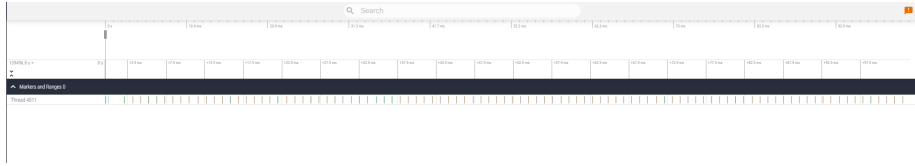


Figure 4: Rocrx Trace Visualization.

3.5 Tracing Control

Tracing control can be achieved through the following ways:

- Filtering by event names using the input file.
- Programatically control the tracing through code annotation.
- Control the flush rate of trace buffer.

Below, we discuss each of these methods in detail.

3.5.1 Filtering by event names

Trace filtering can be achieved by providing the event names in the input file. A sample input file is shown below.

```
$cat input.txt
hsa : hsa_queue_create hsa_amd_memory_pool_allocate
$rocprow -i input.txt --hsa-trace ./MatrixTranspose
$cat outputFolder/rpl.../input.../hsa_api_trace.txt
145896515787851:145896515985201 5209:5209
hsa_amd_memory_pool_allocate({handle=31705344},
1052672, 0, 0x7ffc604ce110) = 0
145896516546789:145896516717998 5209:5209
hsa_amd_memory_pool_allocate({handle=31705344},
1052672, 0, 0x7ffc604ce110) = 0
.....
```

The above input would generate HSA tracing information of these two events only.

3.5.2 Code Annotation

Code annotations are used to describe events in the application. Using roctx APIs, user can insert markers and ranges into the code to describe an event.

3.5.3 Flush rate

The flush rate option can be used to dump the traces periodically. The argument can be in seconds, microseconds, or milliseconds.

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

4 Known Issues

- Concurrent kernel profiling is currently not supported.
- In case of OpenMPI, user need to run rocprof inside mpirun command. If user runs rocprof command before mpirun then tool fails with the following error: "roctracer: Loading 'libamdhip64.so' failed, (null)"
- Currently, GPU id in the HSA visualization is always 0.

5 Errors

The errors are logged to global logs:

```
/tmp/rocprofiler_log.txt  
/tmp/roctracer_log.txt
```

6 Planned features

Following are the features planned in future releases.

- Support for concurrent kernels while profiling.
- A replay engine that lets user supply any number of counters and tool will group them accordingly and re-run the application as many times as needed.

References

- [1] <https://github.com/ROCm-Developer-Tools/roctracer/tree/amd-master/test/MatrixTranspose>.
- [2] <https://perfetto.dev/>.