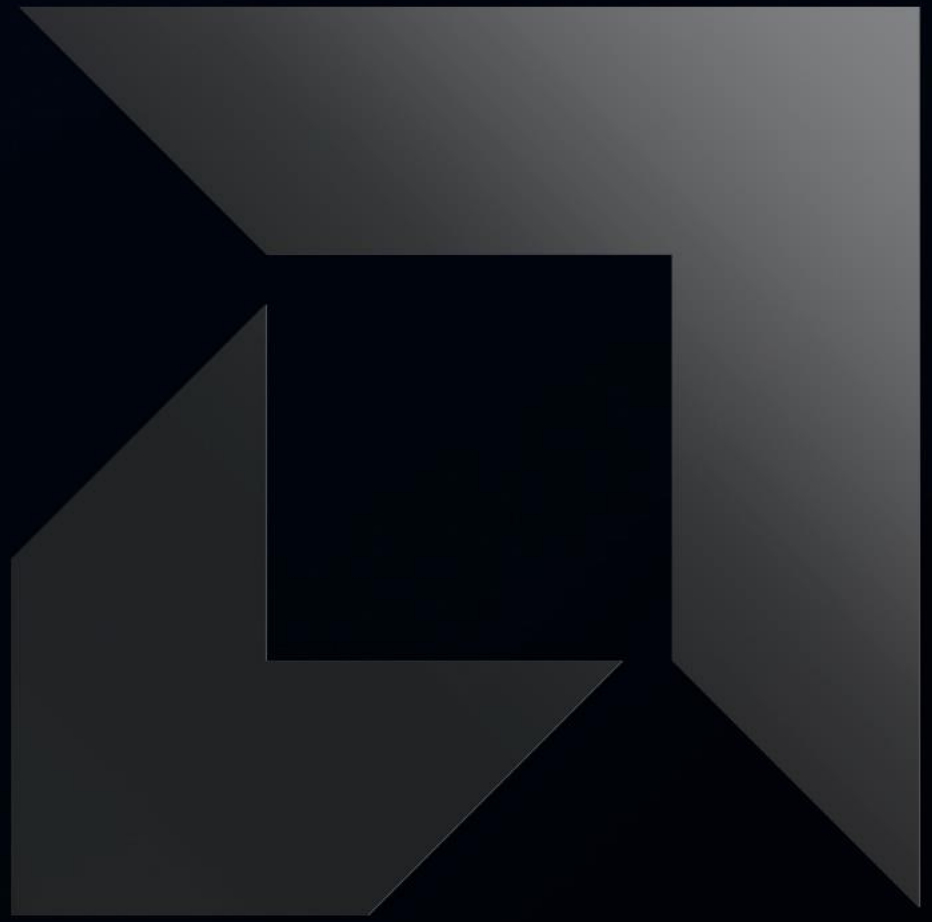


GPU Timeline Profiling

Presenter: Sam Antao

LUMI Pre-hackathon training

May 7th, 2025

AMD

together we advance_

Background – AMD Profilers

ROC-profiler (rocprof)

Hardware Counters	Raw collection of GPU counters and traces
	Counter collection with user input files / Counter results printed to a CSV
Traces and timelines	Trace collection support for
	CPU copy / HIP API / HSA API / GPU Kernels
Visualisation	Traces visualized with Perfetto

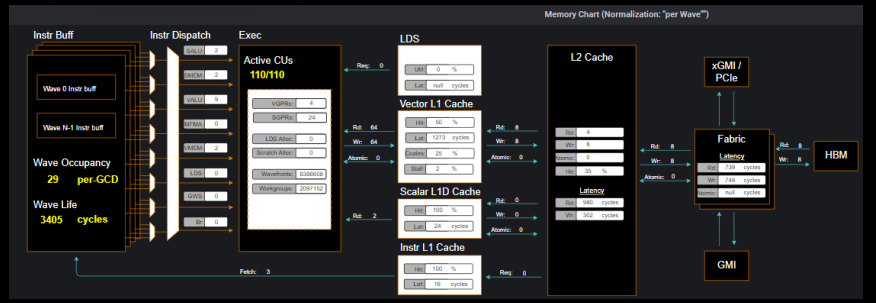
Omnitrace

Trace collection	Comprehensive trace collection
	CPU / GPU
Supports	CPU copy / HIP API / HSA API / GPU Kernels
	OpenMP® / MPI / Kokkos / p-threads / multi-GPU
Visualisation	Traces visualized with Perfetto

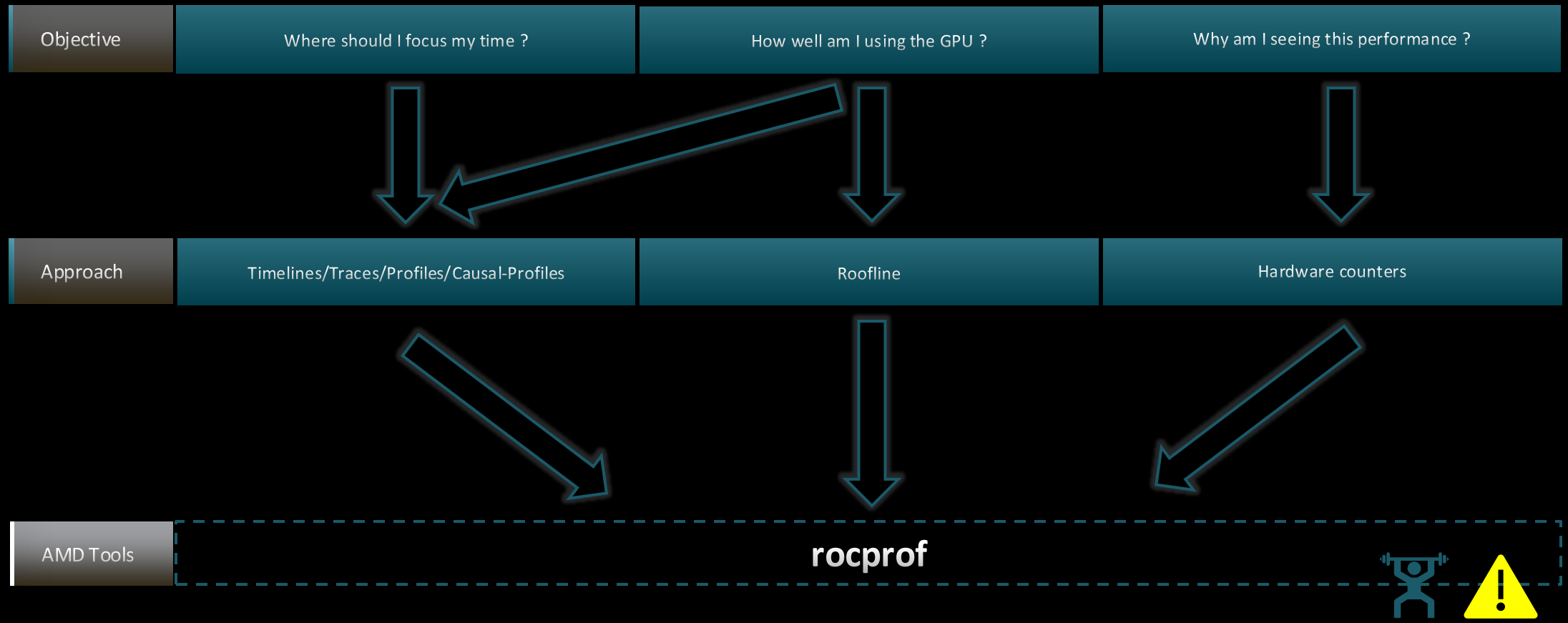
Omniperf

Performance Analysis	Automated collection of hardware counters
	Analysis / Visualisation
Supports	Speed of Light / Memory chart / Rooflines / Kernel comparison
Visualisation	standalone GUI

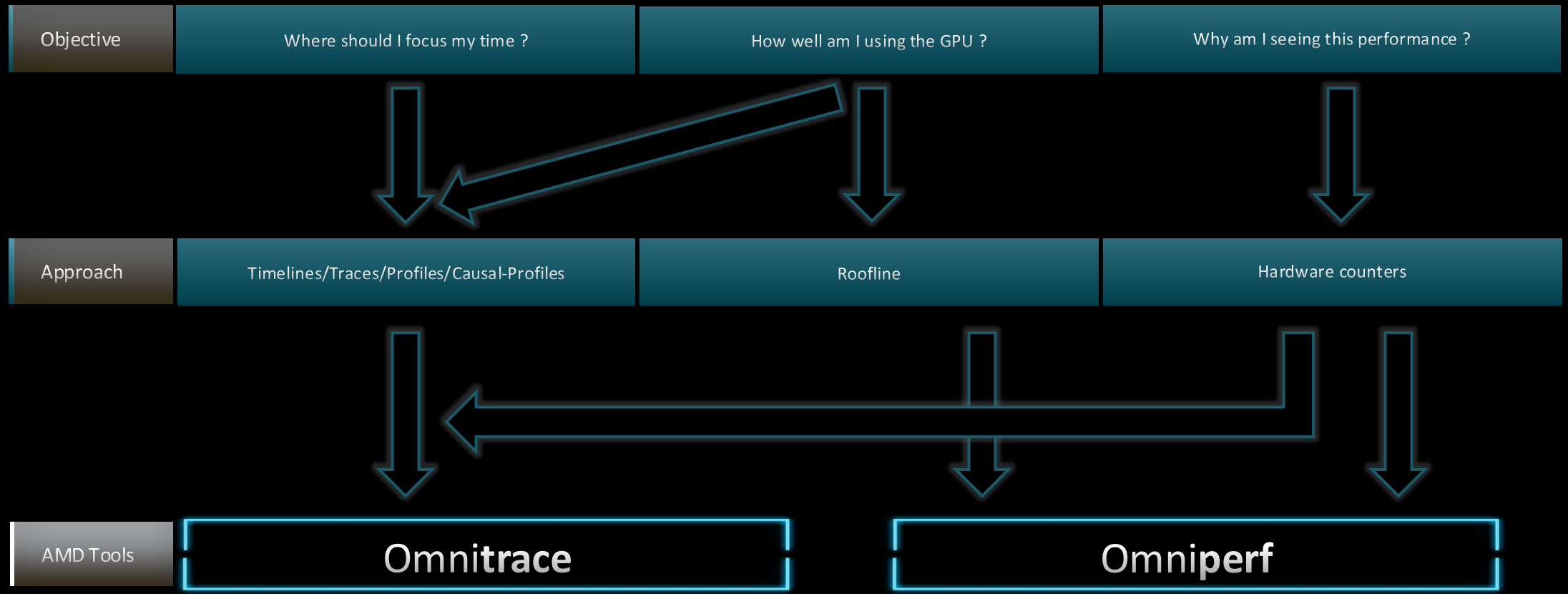
	A	B	C	D	E
1	Name	Calls	TotalDura	AverageNs	Percentage
2	hipMemcpyAsync	99	3.22E+10	3.25E+08	44.14872
3	hipEventSynchronize	330	2.42E+10	73394557	33.225
4	hipMemsetAsync	87	7.76E+09	89232696	10.64953
5	hipHostMalloc	9	5.41E+09	6.01E+08	7.415198
6	hipDeviceSynchronize	28	1.32E+09	47006288	1.805515
7	hipHostFree	17	1.05E+09	61534688	1.435014
8	hipMemcpy	41	8.11E+08	19791876	1.113161
9	hipLaunchKernel	1856	58082083	31294	0.079676
10	hipStreamCreate	2	46380834	23190417	0.063625
11	hipMemset	2	18847246	9423623	0.025854
12	hipStreamDestroy	2	15183338	7591669	0.020828
13	hipFree	38	8269713	217624	0.011344
14	hipEventRecord	330	2520035	7636	0.003457
15	hipMalloc	30	1484804	49493	0.002037
16	__hipPopCallConfigura	1856	229159	123	0.000314
17	__hipPushCallConfigur	1856	224177	120	0.000308
18	hipGetLastError	1494	100458	67	0.000138
19	hipEventCreate	330	76675	232	0.000105
20	hipEventDestroy	330	64671	195	8.87E-05
21	hipGetDevicePropertie	47	51808	1102	7.11E-05
22	hipGetDevice	64	11611	181	1.59E-05
23	hipSetDevice	1	401	401	5.50E-07
24	hipGetDeviceCount	1	220	220	3.02E-07



Background – AMD Profilers



Background – AMD Profilers



Introduction to ROC-Profiler

Presenter: Sam Antao

LUMI Pre-hackathon training

October 8th , 2024

AMD

together we advance_

ROCM ON LUMI

Meant to support older version of apps and frameworks

Facilitate transition

GPU address sanitizer (beta)

Data pre-processing capabilities (MIVisionX)

GPU-Aware MPI

Default version

Officially supported

Recommended for debugging

Improved sparse matrix operations

Latest Pytorch and other AI frameworks require this version

Introduced many performance improvements

ROCPROF V3

Many stability and performance improvements for performance libraries

Improved support for lower precisions

Best tunned for AI inference workloads

Integration of profiling tools
Autocast (mixed-precision)

Native OpenXLA support

	Dec 2023	Mar 2024	Jun 2024	Aug 2024	Nov 2024	Apr 2025
User level	5.7.3	**6.0.3**	6.1.3	6.2	6.3 ?	6.4 ?
Driver		**6.0.3**			We'll likely be abusing the driver soon	

What is ROC-Profiler (v1-v2-v3)?

- ROC-profiler (also referred to as `rocprof`) is the command line front-end for AMD's GPU profiling libraries
  - Repo: https://github.com/ROCm-Developer-Tools/rocprofiler
- rocprof contains the central components allowing application traces and counter collection
  - Under constant development
- Distributed with ROCm
- The output of `rocprofv1` can be visualized in the Chrome browser with Perfetto (https://ui.perfetto.dev/)
- There are ROCProfiler V1 and V2 (roctracer and rocprofiler into single library, same API)
- ROC-profiler-SDK is a profiling and tracing library for HIP and ROCm application. The new API improved thread safety and includes more efficient implementations and provides a tool library to support on writing your tool implementations. It is still in beta release.
- `rocprofv3` uses this tool library to profile and trace applications.

rocprof (v1): Getting Started + Useful Flags

- To get help:
  `${ROCM_PATH}/bin/rocprof -h`
- Useful housekeeping flags:
  - `--timestamp <on|off>` - turn on/off gpu kernel timestamps
  - `--basenames <on|off>` - turn on/off truncating gpu kernel names (i.e., removing template parameters and argument types)
  - `-o <output csv file>` - Direct counter information to a particular file name
  - `-d <data directory>` - Send profiling data to a particular directory
  - `-t <temporary directory>` - Change the directory where data files typically created in /tmp are placed. This allows you to save these temporary files.
- Flags directing rocprofiler activity:
  - `-i input<.txt|.xml>` - specify an input file (note the output files will now be named input.*)
  - `--hsa-trace` - to trace GPU Kernels, host HSA events (more later) and HIP memory copies.
  - `--hip-trace` - to trace HIP API calls
  - `--roctx-trace` - to trace roctx markers
  - `--kfd-trace` - to trace GPU driver calls
- Advanced usage
  - `-m <metric file>` - Allows the user to define and collect custom metrics. See rocprofiler/test/tool/*.xml on GitHub for examples.

rocprof (v1): : Kernel Information

- rocprof can collect kernel(s) execution stats

```
$ /opt/rocm/bin/rocprof --stats --basenames on <app with arguments>
```

- This will output two csv files:
  - `results.csv:` information per each call of the kernel
  - `results.stats.csv:` statistics grouped by each kernel
- Content of results.stats.csv to see the list of GPU kernels with their durations and percentage of total GPU time:

```
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"JacobiIterationKernel",1000,556699359,556699,43.291753895270446
"NormKernel1",1001,430797387,430367,33.500980655394606
"LocalLaplacianKernel",1000,280014065,280014,21.775307970480817
"HaloLaplacianKernel",1000,14635177,14635,1.1381052818810995
"NormKernel2",1001,3770718,3766,0.2932300765671734
"__amd_rocclr_fillBufferAligned.kd",1,8000,8000,0.0006221204058583505
```

- In a spreadsheet viewer, it is easier to read:

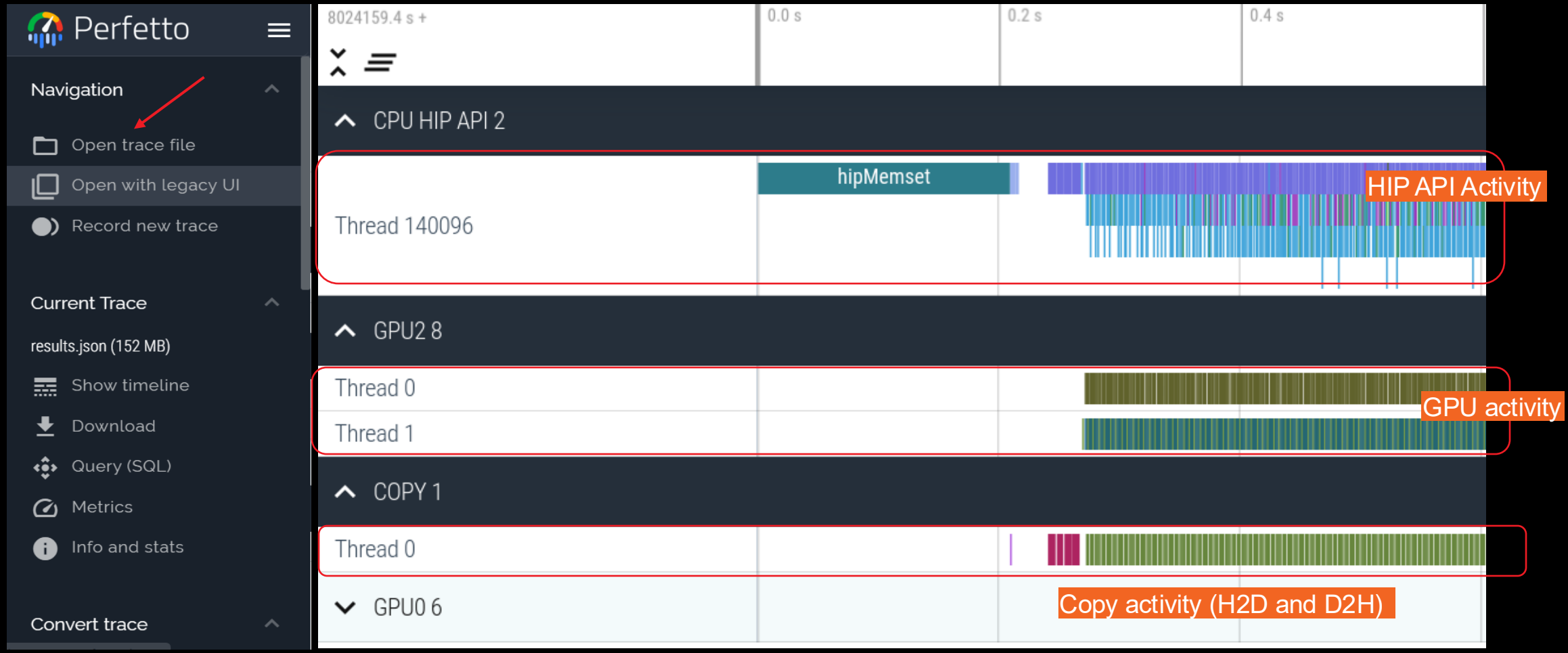
	A	B	C	D	E
1	Name	Calls	TotalDurationNs	AverageNs	Percentage
2	JacobiIterationKernel	1000	556699359	556699	43.2917538952704
3	NormKernel1	1001	430797387	430367	33.5009806553946
4	LocalLaplacianKernel	1000	280014065	280014	21.7753079704808
5	HaloLaplacianKernel	1000	14635177	14635	1.1381052818811
6	NormKernel2	1001	3770718	3766	0.293230076567173
7	__amd_rocclr_fillBufferAligned	1	8000	8000	0.000622120405858

rocprof (v1): + Perfetto: Collecting and Visualizing App Traces

- rocprof can collect traces

  ```
  $ /opt/rocm/bin/rocprof --hip-trace <app with arguments>
  ```

  This will output a .json file that can be visualized using the Chrome browser and Perfetto (https://ui.perfetto.dev/)



rocprofv3: Getting Started + Useful Flags

- To get help:
  `${ROCM_PATH}/bin/rocprofv3 -h`
- Useful housekeeping flags:
  - `--hip-trace` For Collecting HIP Traces (runtime + compiler)
  - `--hip-runtime-trace` For Collecting HIP Runtime API Traces
  - `--hip-compiler-trace` For Collecting HIP Compiler generated code Traces
  - `--marker-trace` For Collecting Marker (ROCTx) Traces
  - `--memory-copy-trace` For Collecting Memory Copy Traces
  - `--stats` For Collecting statistics of enabled tracing types
  - `--hsa-trace` For Collecting HSA Traces (core + amd + image + finalizer)
  - `--hsa-core-trace` For Collecting HSA API Traces (core API)
  - `--hsa-amd-trace` For Collecting HSA API Traces (AMD-extension API)
  - `--hsa-image-trace` For Collecting HSA API Traces (Image-extenson API)
  - `--hsa-finalizer-trace` For Collecting HSA API Traces (Finalizer-extension API)

rocprofv3: Getting Started + Useful Flags (II)

- Useful housekeeping flags:
  - -s, --sys-trace For Collecting HIP, HSA, Marker (ROCTx), Memory copy, Scratch memory, and Kernel dispatch traces
  - -M, --mangled-kernels Do not demangle the kernel names
  - -T, --truncate-kernels Truncate the demangled kernel names
  - -L, --list-metrics List metrics for counter collection
  - -i INPUT, --input INPUT Input file for counter collection
  - -o OUTPUT_FILE, --output-file OUTPUT_FILE
    For the output file name
  - -d OUTPUT_DIRECTORY, --output-directory OUTPUT_DIRECTORY
    For adding output path where the output files will be saved
  - --output-format {csv,json,pftrace} [{csv,json,pftrace} ...]
    For adding output format (supported formats: csv, json, pftrace)
  - --log-level {fatal,error,warning,info,trace}
    Set the log level
  - --kernel-names KERNEL_NAMES [KERNEL_NAMES ...]
    Filter kernel names
  - --preload [PRELOAD ...]
    Libraries to prepend to LD_PRELOAD (usually for sanitizers)

  - rocprofv3 requires double-hyphen (--) before the application to be executed, e.g.

```
$ rocprofv3 [<rocprofv3-option> ...] -- <application> [<application-arg> ...]
$ rocprofv3 --hip-trace -- ./myapp -n 1
```

  - Instructions: https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/docs-6.2.1/how-to/using-rocprofv3.html

rocprofv3: Kernel Information

- rocprof can collect kernel(s) execution stats

```
$ /opt/rocm/bin/rocprofv3 --stats --kernel-trace -T -- <app with arguments>
```

- This will output four csv files (XXXXX are numbers):
  - XXXXX_agent_info.csv: information for the used hardware APU/GPU and CPU
  - XXXXX_kernel_traces.csv: information per each call of the kernel
  - XXXXX_kernel_stats.csv: statistics grouped by each kernel
  - XXXXX_domain_stats.csv: statistics grouped by domain, such as KERNEL_DISPATCH, HIP_COMPILER_API
- Content of results.stats.csv to see the list of GPU kernels with their durations and percentage of total GPU time:

```
"Name","Calls","TotalDurationNs","AverageNs","Percentage","MinNs","MaxNs","StdDev"
"NormKernel1",1001,365858158,365492.665335,53.49,360561,449240,3460.551681
"JacobiIterationKernel",1000,171479968,171479.968000,25.07,162040,205241,10113.842491
"LocalLaplacianKernel",1000,135771713,135771.713000,19.85,130400,145121,3349.580100
"HaloLaplacianKernel",1000,7777189,7777.189000,1.14,7000,12120,349.399610
"NormKernel2",1001,3107927,3104.822178,0.4544,2200,138681,6466.048652
"__amd_rocclr_fillBufferAligned",1,2720,2720.000000,3.977e-04,2720,2720,0.00000000e+00
```

- In a spreadsheet viewer, it is easier to read:

	A	B	C	D	E	F	G	H
1	Name	Calls	TotalDurationNs	AverageNs	Percentage	MinNs	MaxNs	StdDev
2	NormKernel1	1001	365858158	365492.665	53.49	360561	449240	3460.552
3	JacobiIterationKernel	1000	171479968	171479.968	25.07	162040	205241	10113.84
4	LocalLaplacianKernel	1000	135771713	135771.713	19.85	130400	145121	3349.58
5	HaloLaplacianKernel	1000	7777189	7777.189	1.14	7000	12120	349.3996
6	NormKernel2	1001	3107927	3104.82218	0.4544	2200	138681	6466.049
7	__amd_rocclr_fillBufferAligned	1	2720	2720	3.98E-04	2720	2720	0

rocprofv3: Collecting Application Traces

- rocprof can collect a variety of trace event types, and generate timelines in JSON format for use with Perfetto, currently, however better use the pftrace output format (*--output-format pftrace*):

Trace Event	rocprof Trace Mode
HIP API call	`--hip-trace`
GPU Kernels	`--kernel-trace`
Host <-> Device Memory copies	`--hip-trace or --memory-copy-trace`
CPU HSA Calls	`--hsa-trace`
User code markers	`--marker-trace`
Collect HIP, HSA, Kernels, Memory Copy, Marker API	`--sys-trace`
Scratch memory operations	`--scratch-memory-trace`

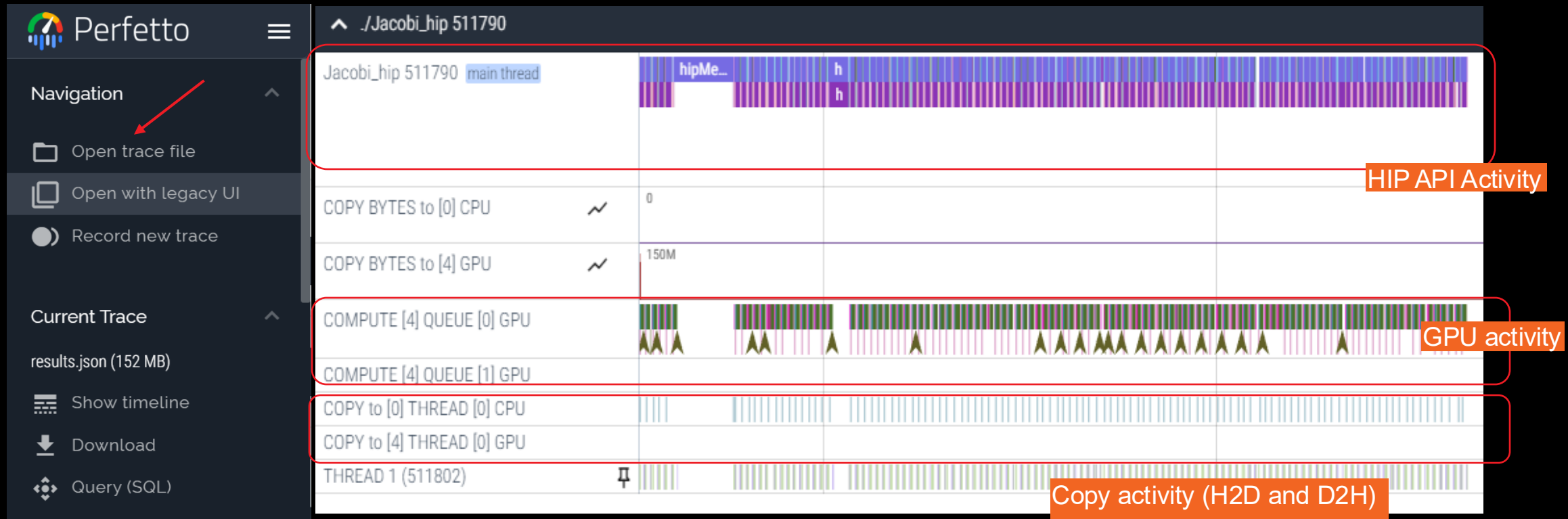
- You can combine modes like --stats `--hip-trace --hsa-trace --output-format pftrace`

rocprof + Perfetto: Collecting and Visualizing Application Traces

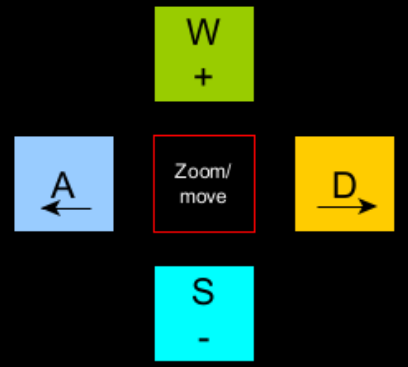
- rocprof can collect traces

  ```
  $ /opt/rocm/bin/rocprof --hip-trace --output-format pftrace -- <app with arguments>
  ```

  This will output a pftrace file that can be visualized using the chrome browser and Perfetto (https://ui.perfetto.dev/)



Perfetto: Visualizing Application Traces

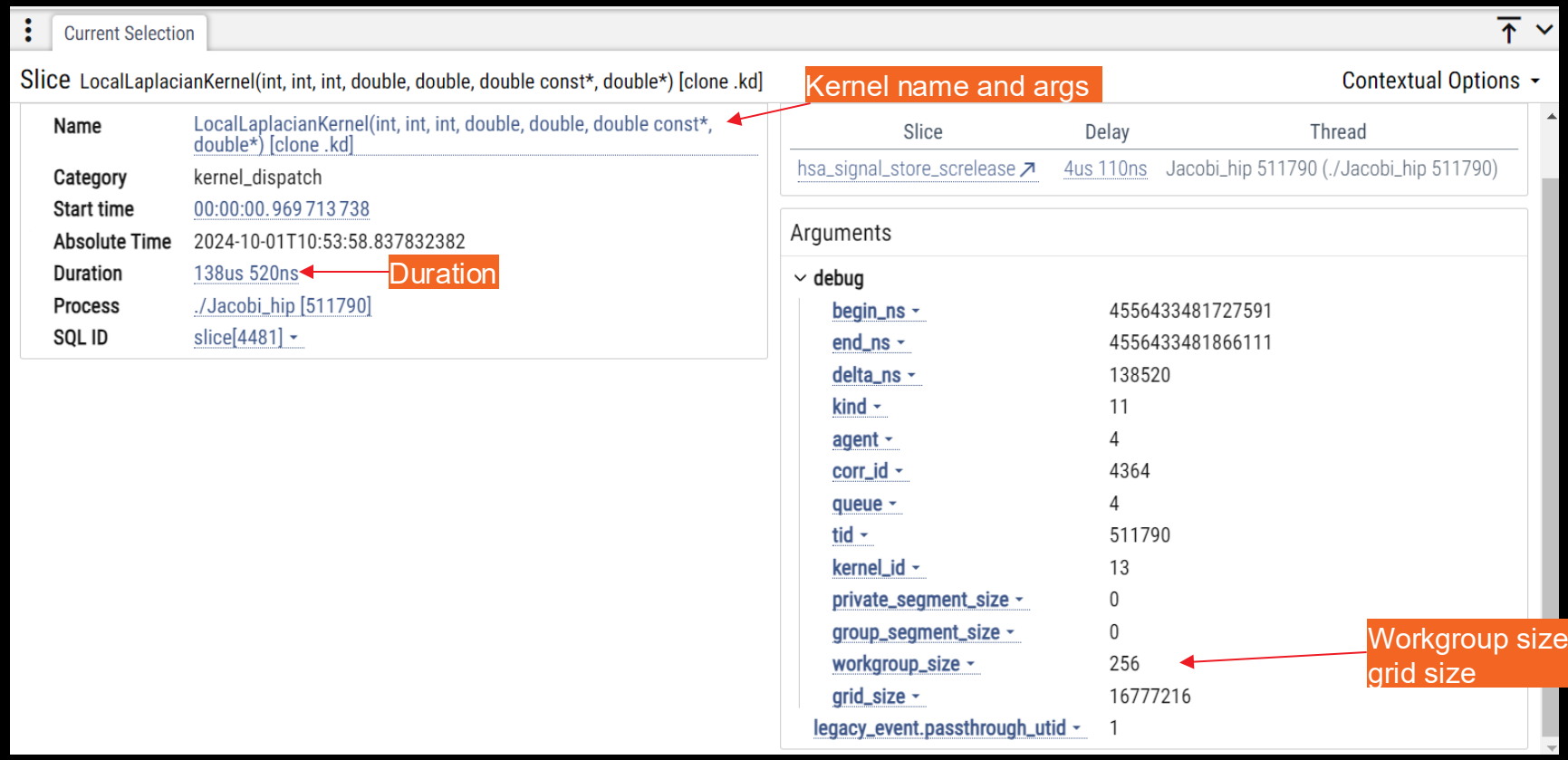


- Zoom in to see individual events
- Navigate trace using WASD keys

Perfetto: Kernel Information and Flow Events

- Zoom and select a kernel, you can see the link to the HIP call launching the kernel
- Try to open the information for the kernel (button at bottom right)

Perfetto: Kernel Information



Rocprofv3: OpenMP Offloading

- The option --kernel-trace provides information of the OpenMP kernels, good to use --hsa-trace if you want information from HSA layer

- For example:

srun -n 1 rocprofv3 --stats --kernel-trace --output-format pftrace -- <app with arguments>

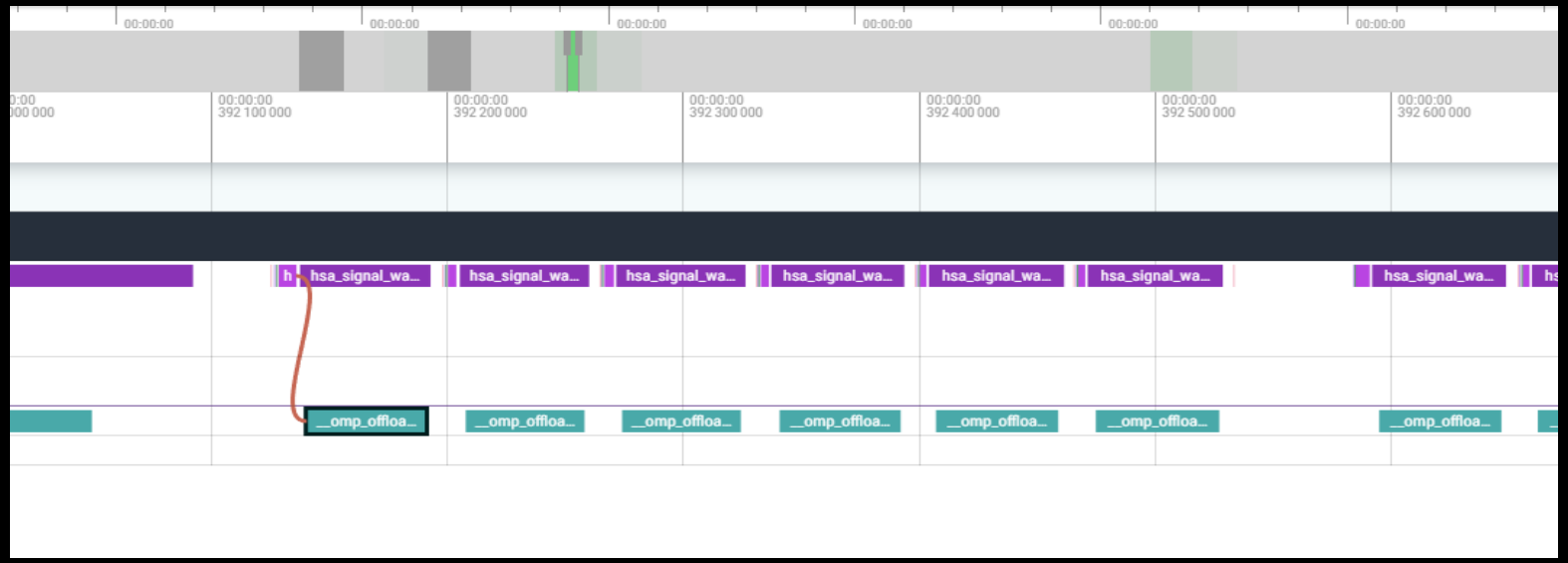
Content of XXXXX_kernel_stats.csv:

```
"Name","Calls","TotalDurationNs","AverageNs","Percentage","MinNs","MaxNs","StdDev"
"__omp_offloading_32_7f7a__Z6evolveR5FieldS0_dd_l24",500,45818062,91636.124000,100.00,49840,19483408,868965.767084
```

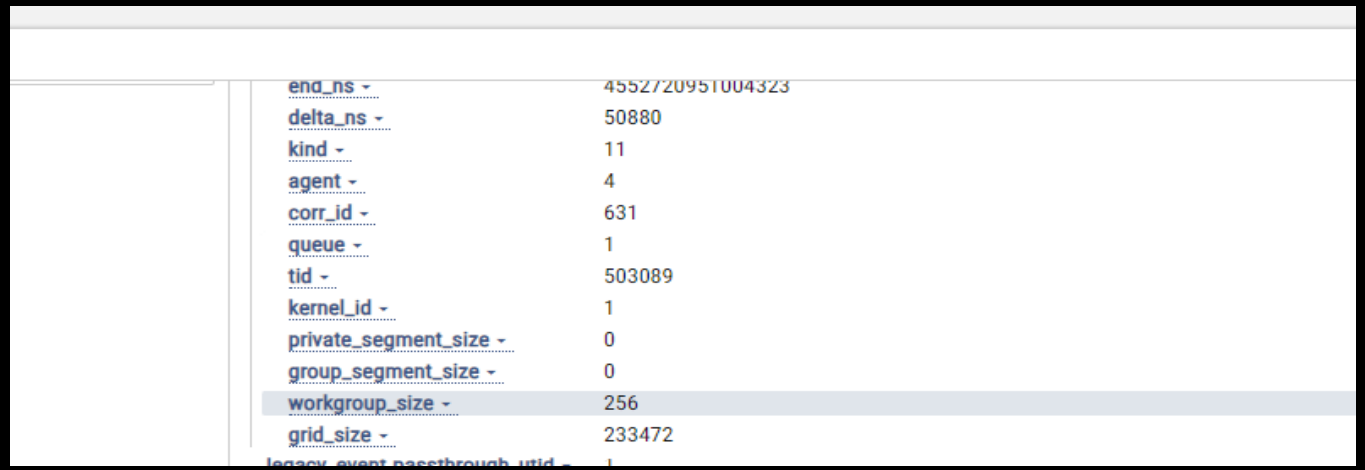
Content of XXXXX_kernel_trace.csv

```
"Kind","Agent_Id","Queue_Id","Kernel_Id","Kernel_Name","Correlation_Id","Start_Timestamp","End_Timestamp","Private_Segment_Size","Group_Segment_Size","
Workgroup_Size_X","Workgroup_Size_Y","Workgroup_Size_Z","Grid_Size_X","Grid_Size_Y","Grid_Size_Z"
"KERNEL_DISPATCH",4,1,1,"__omp_offloading_32_7f7a__Z6evolveR5FieldS0_dd_l24",1,4547852833814530,4547852853297938,0,0,256,1,1,233472,1,1
"KERNEL_DISPATCH",4,1,1,"__omp_offloading_32_7f7a__Z6evolveR5FieldS0_dd_l24",2,4547852853393869,4547852853446789,0,0,256,1,1,233472,1,1
"KERNEL_DISPATCH",4,1,1,"__omp_offloading_32_7f7a__Z6evolveR5FieldS0_dd_l24",3,4547852853461519,4547852853514599,0,0,256,1,1,233472,1,1
…
```

Perfetto and OpenMP visualization



- Using: *--sys-trace --output-format pftrace*
- We can use: *--kernel-trace --output-format pftrace*



rocprofv3: Collecting Application Traces with rocTX Markers and Regions

- rocprofv3 can collect user defined regions or markers using rocTX
- Annotate code with roctx regions:

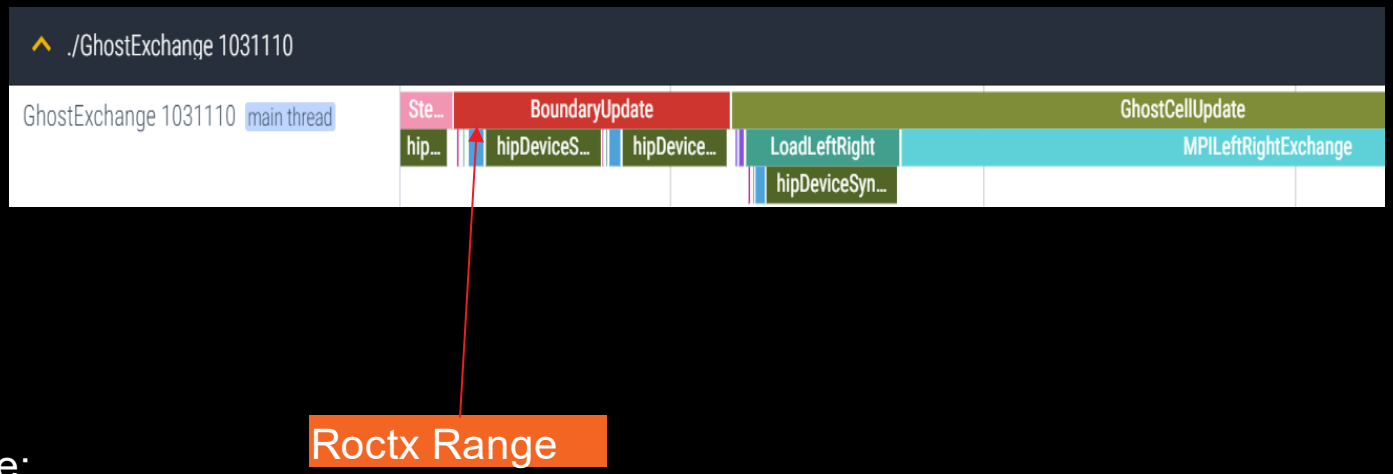
```
#include <rocprofiler-sdk-roctx/roctx.h>
...
    roctxRangePush("reduce_for_c");
    reduce_function ();
    roctxRangePop();
...
```

- Annotate code with roctx markers:

```
...
    roctxMark("start of some code");
    // some_code
    roctxMark("end of some code");
...
```

- Add roctx and roctracer libraries to link line:

```
-L${ROCM_PATH}/lib -lrocprofiler-sdk-roctx -lroctracer64
```



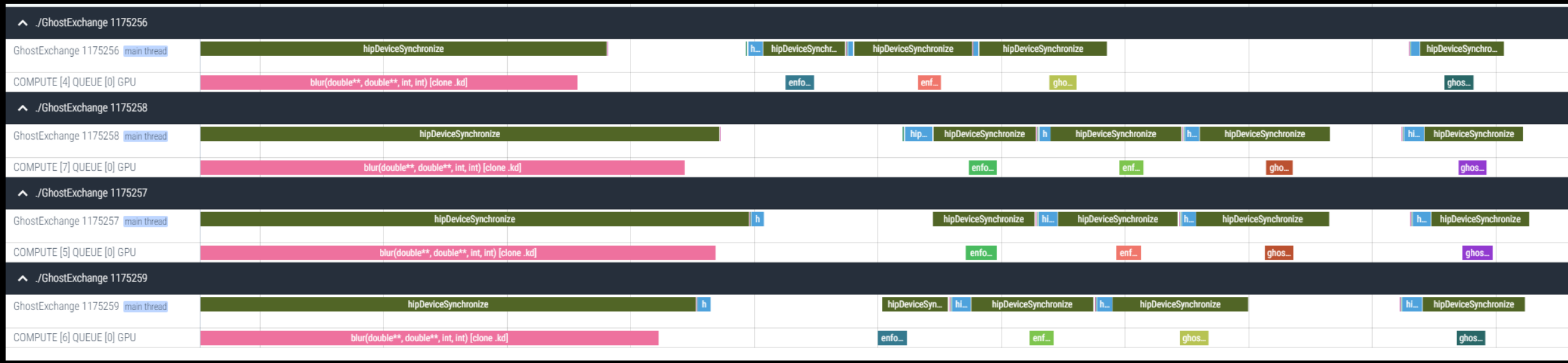
- Profile with `--roctx-range` option:

```
$ /opt/rocm/bin/rocprofv3 --hip-trace --marker-trace -- <app with arguments>
```

- Important: There is some difference regarding roctx between rocprof and rocprofv3

Rocprofv3: Merge traces

- When you have one pftrace per MPI processes you can merge them as follows:
  - For example cat XXXXX_results.pftrace > all_ghostexchange.pftrace
  - Then visualize the file called all_ghostexchange.pftrace



rocprofv3: Commonly Used GPU Counters

VALUUtilization	The percentage of ALUs active in a wave. Low VALUUtilization is likely due to high divergence or a poorly sized grid
VALUBusy	The percentage of GPUTime vector ALU instructions are processed. Can be thought of as something like compute utilization
FetchSize	The total kilobytes fetched from global memory
WriteSize	The total kilobytes written to global memory
MemUnitStalled	The percentage of GPUTime the memory unit is stalled
CU_OCCUPANCY	The ratio of active waves on a CU to the maximum number of active waves supported by the CU
MeanOccupancyPerCU	Mean occupancy per compute unit
MeanOccupancyPerActiveCU	Mean occupancy per active compute unit

Full list at: https://github.com/ROCm-Developer-Tools/rocprofiler/blob/amd-master/test/tool/metrics.xml

rocprofv3: Collecting Hardware Counters

- rocprofv3 can collect a number of hardware counters and derived counters
  - `$ /opt/rocm/bin/rocprofv3 -L`

- Specify counters in a counter file. For example:
  - `$ /opt/rocm/bin/rocprofv3 -i rocprof_counters.txt -- <app with args>`
  - `$ cat rocprof_counters.txt`

    ```
    pmc: VALUUtilization VALUBusy FetchSize WriteSize MemUnitStalled
    pmc: GPU_UTIL CU_OCCUPANCY MeanOccupancyPerCU MeanOccupancyPerActiveCU
    ```

  - A limited number of counters can be collected during a specific pass of code
    - Each line in the counter file will be collected in one pass
    - You will receive an error suggesting alternative counter ordering if you have too many / conflicting counters on one line

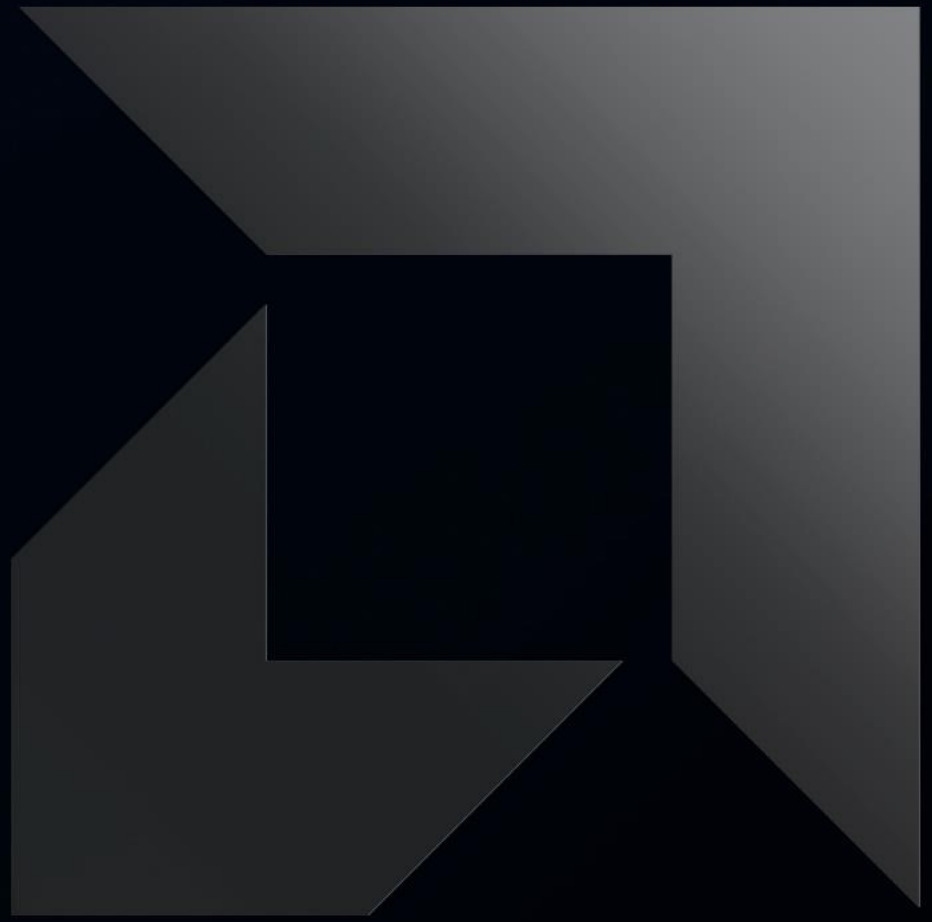
  - One directory per pmc line will be created, for example pmc_1 and pmc_2 for the two lines in the file with the counters.
  - One agent_info and one counter_collection csv file per MPI process will be created containing all the requested counters for each invocation of every kernel

rocprof: Profiling Overhead

- As with every profiling tool, there is an overhead
- The percentage of the overhead depends on the profiling options used
  - For example, tracing is faster than hardware counter collection
- When collecting many counters, the collection may require multiple passes
- With rocTX markers/regions, tracing can take longer and the output may be large
  - Sometimes too large to visualize
- The more data collected, the more the overhead of profiling
  - Depends on the application and options used
- rocprofv3 has less overhead than rocprof (v1) on various examples with extensive ROCm calls

Summary

- rocprof is the open source, command line AMD GPU profiling tool distributed with rocprofv3 available from ROCm 6.2 and later
- rocprof provides tracing of GPU kernels, through various options, HIP API, HSA API, Copy activity and others
- rocprof can be used to collect GPU hardware counters with additional overhead
- Perfetto seems to visualize pftrace files without significant issues
- Other output files are in text/CSV format



System Profiling with Omnitrace

Presenter: Sam Antao

LUMI Pre-hackathon training

May 7th, 2025

AMD

together we advance_

AMD Profilers with Timeline Profiling Support

ROC-profiler (rocprof)

Hardware Counters

- Raw collection of GPU counters and traces
  - Counter collection with user input files
  - Counter results printed to a CSV

Traces and timelines

- Trace collection support for
  - CPU to GPU copy
  - HIP API
  - HSA API
  - GPU Kernels

Omnitrace

Trace collection

- Hardware counters (rocprof)
- Sampling
- Runtime instrumentation with standard executable
- Binary rewrite Pre-instruments executable

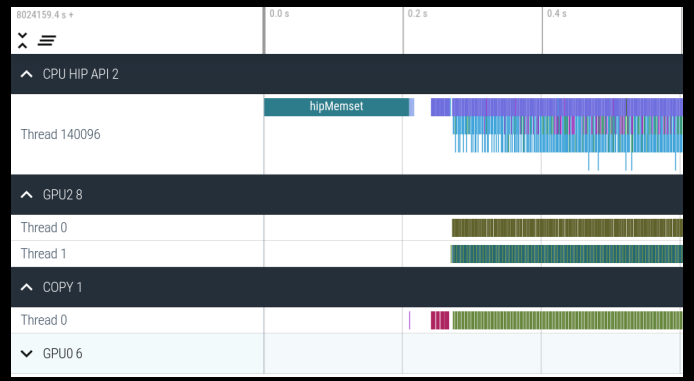
Adds to rocprof

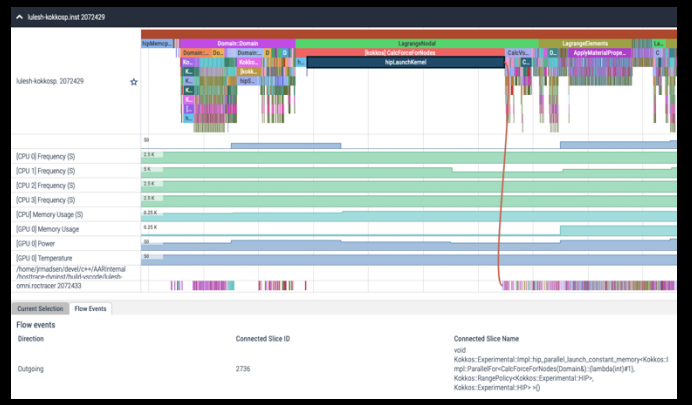
- PAPI
- DynInst
- Perf
- Profile
- AMDμProf
- OpenMP®
- MPI
- Kokkos
- p-threads
- rocm-smi

Visualization

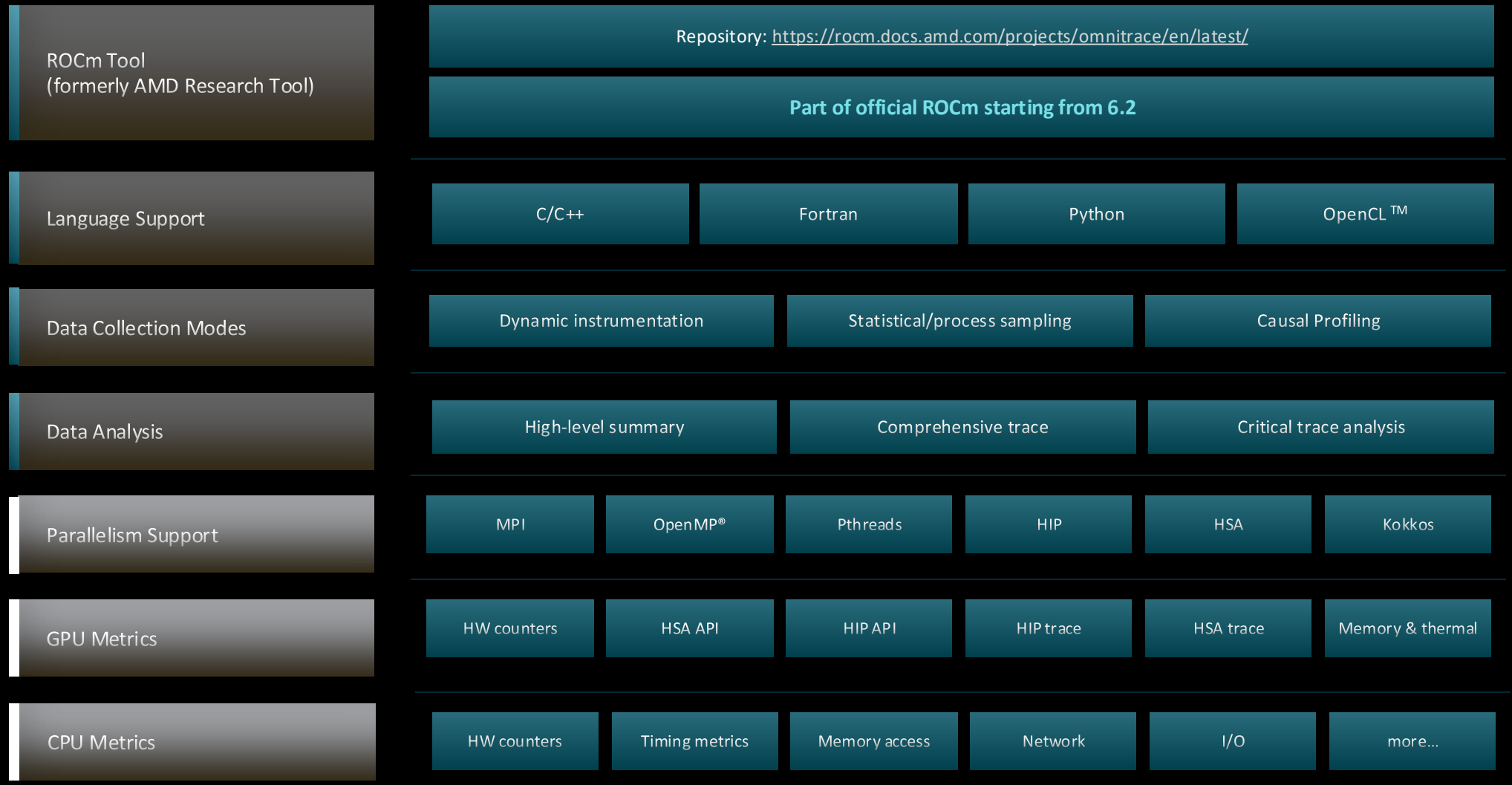
Traces visualized with Perfetto

	A	B	C	D	E
1	Name	Calls	TotalDura	AverageN	Percentage
2	hipMemcpyAsync	99	3.22E+10	3.25E+08	44.14872
3	hipEventSynchronize	330	2.42E+10	73394557	33.225
4	hipMemsetAsync	87	7.76E+09	89232696	10.64953
5	hipHostMalloc	9	5.41E+09	6.01E+08	7.415198
6	hipDeviceSynchronize	28	1.32E+09	47006288	1.805515
7	hipHostFree	17	1.05E+09	61534688	1.435014
8	hipMemcpy	41	8.11E+08	19791876	1.113161
9	hipLaunchKernel	1856	58082083	31294	0.079676
10	hipStreamCreate	2	46380834	23190417	0.063625
11	hipMemset	2	18847246	9423623	0.025854
12	hipStreamDestroy	2	15183338	7591669	0.020828
13	hipFree	38	8269713	217624	0.011344
14	hipEventRecord	330	2520035	7636	0.003457
15	hipMalloc	30	1484804	49493	0.002037
16	__hipPopCallConfigura	1856	229159	123	0.000314
17	__hipPushCallConfigur	1856	224177	120	0.000308
18	hipGetLastError	1494	100458	67	0.000138
19	hipEventCreate	330	76675	232	0.000105
20	hipEventDestroy	330	64671	195	8.87E-05
21	hipGetDevicePropertie	47	51808	1102	7.11E-05
22	hipGetDevice	64	11611	181	1.59E-05
23	hipSetDevice	1	401	401	5.50E-07
24	hipGetDeviceCount	1	220	220	3.02E-07





Omnitrace: Application Profiling, Tracing, and Analysis



New features constantly being added

Refer to current documentation for recent updates

Omnitrace Configuration File

```
$ omnitrace-avail --categories [options]
```

Get more information about run-time settings, data collection capabilities, and available hardware counters. For more information or help use `-h/--help` flags:

```
$ omnitrace-avail -h
```

Collect information for omnitrace-related settings using shorthand `-c` for `--categories`:

```
$ omnitrace-avail -c omnitrace
```

For brief description, use the options:

```
$ omnitrace-avail -bd
```

ENVIRONMENT VARIABLE	DESCRIPTION
OMNITRACE_CAUSAL_BINARY_EXCLUDE	Excludes binaries matching the list of provided regexes from causal experiments (separated by tab, sem...
OMNITRACE_CAUSAL_BINARY_SCOPE	Limits causal experiments to the binaries matching the provided list of regular expressions (separated...
OMNITRACE_CAUSAL_DELAY	Length of time to wait (in seconds) before starting the first causal experiment
OMNITRACE_CAUSAL_DURATION	Length of time to perform causal experimentation (in seconds) after the first experiment has started. ...
OMNITRACE_CAUSAL_FUNCTION_EXCLUDE	Excludes functions matching the list of provided regexes from causal experiments (separated by tab, se...
OMNITRACE_CAUSAL_FUNCTION_SCOPE	List of <function> regex entries for causal profiling (separated by tab, semi-colon, and/or quotes (si...
OMNITRACE_CAUSAL_RANDOM_SEED	Seed for random number generator which selects speedups and experiments -- please note that the lines ...
OMNITRACE_CAUSAL_SOURCE_EXCLUDE	Excludes source files or source file + lineno pair (i.e. <file> or <file>:<line>) matching the list of...
OMNITRACE_CAUSAL_SOURCE_SCOPE	Limits causal experiments to the source files or source file + lineno pair (i.e. <file> or <file>:<lin...
OMNITRACE_CONFIG_FILE	Configuration file for omnitrace
OMNITRACE_CRITICAL_TRACE	Enable generation of the critical trace
OMNITRACE_ENABLED	Activation state of timemory
OMNITRACE_OUTPUT_PATH	Explicitly specify the output folder for results
OMNITRACE_OUTPUT_PREFIX	Explicitly specify a prefix for all output files
OMNITRACE_PAPI_EVENTS	PAPI presets and events to collect (see also: papi_avail)
OMNITRACE_PERFETTO_BACKEND	Specify the perfetto backend to activate. Options are: 'inprocess', 'system', or 'all'
OMNITRACE_PERFETTO_BUFFER_SIZE_KB	Size of perfetto buffer (in KB)
OMNITRACE_PERFETTO_FILL_POLICY	Behavior when perfetto buffer is full. 'discard' will ignore new entries, 'ring_buffer' will overwrite...
OMNITRACE_PROCESS_SAMPLING_DURATION	If > 0.0, time (in seconds) to sample before stopping. If less than zero, uses OMNITRACE_SAMPLING_DURA...
OMNITRACE_PROCESS_SAMPLING_FREQ	Number of measurements per second when OMNITTRACE_USE_PROCESS_SAMPLING=ON. If set to zero, uses OMNITR...
OMNITRACE_ROCM_EVENTS	ROCm hardware counters. Use ':device=N' syntax to specify collection on device number N, e.g. ':device...
OMNITRACE_SAMPLING_CPUS	CPUs to collect frequency information for. Values should be separated by commas and can be explicit or...
OMNITRACE_SAMPLING_DELAY	Time (in seconds) to wait before the first sampling signal is delivered, increasing this value can fix...
OMNITRACE_SAMPLING_DURATION	If > 0.0, time (in seconds) to sample before stopping
OMNITRACE_SAMPLING_FREQ	Number of software interrupts per second when OMNITTRACE_USE_SAMPLING=ON
OMNITRACE_SAMPLING_GPUS	Devices to query when OMNITRACE_USE_ROCM_SMI=ON. Values should be separated by commas and can be expli...

Create a config file

Create a config file in `$HOME`:

```
$ omnitrace-avail -G $HOME/.omnitrace.cfg
```

To add description of all variables and settings, use:

```
$ omnitrace-avail -G $HOME/.omnitrace.cfg --all
```

Modify the config file `$HOME/.omnitrace.cfg` as desired to enable and change settings:

```
<snip>
OMNITRACE_TRACE                                 = true
OMNITRACE_PROFILE                               = true
OMNITRACE_USE_SAMPLING                          = false
OMNITRACE_USE_ROCTRACER                         = true
OMNITRACE_USE_ROCM_SMI                          = true
OMNITRACE_USE_MPIP                              = true
OMNITRACE_USE_PID                               = true
OMNITRACE_USE_ROCPROFILER                       = true
OMNITRACE_USE_ROCTX                             = true
<snip>
```

Contents of the config file

Declare which config file to use by setting the environment:

```
$ export OMNITRACE_CONFIG_FILE=/path-to/.omnitrace.cfg
```

Binary Rewrite

Binary Rewrite

```
$ omnitrace-instrument [omnitrace-options] -o <new-name-of-exec> -- <CMD> <ARGS>
```

Generating a new executable/library with instrumentation built-in:

```
$ omnitrace-instrument -o Jacobi_hip.inst -- ./Jacobi_hip
```

This new binary will have instrumented functions

Subroutine Instrumentation

Default instrumentation is main function and functions of 1024 instructions and more (for CPU)

To instrument routines with 500 or more cycles, add option "`-i 500`" (more overhead)

```
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libgcc_s-8-20210514.so.1'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libnss_compat-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libnss_dns-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libnss_files-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libpthread-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libresolv-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/librt-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libstdc++.so.6.0.25'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libthread_db-1.0.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libutil-2.28.so'...
[omnitrace][exe] [internal] parsing library: '/usr/lib64/libz.so.1.2.11'...
[omnitrace][exe] [internal] binary info processing required 0.666 sec and 110.500 MB
[omnitrace][exe] Processing 9 modules...
[omnitrace][exe] Processing 9 modules... Done (0.001 sec, 0.000 MB)
[omnitrace][exe] Found 'MPI_Init' in '/home/ssitaram/git/HPCTrainingExamples/HIP/jacobi/Jacobi_hip'. Enabling MPI support...
[omnitrace][exe] Finding instrumentation functions...
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/available.json'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/available.txt'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/instrumented.json'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/instrumented.txt'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/excluded.json'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/excluded.txt'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/overlapping.json'... Done
[omnitrace][exe] Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_12.57/instrumentation/overlapping.txt'... Done
[omnitrace][exe]
[omnitrace][exe] The instrumented executable image is stored in '/home/ssitaram/git/HPCTrainingExamples/HIP/jacobi/Jacobi_hip.inst'
[omnitrace][exe] Getting linked libraries for /home/ssitaram/git/HPCTrainingExamples/HIP/jacobi/Jacobi_hip...
[omnitrace][exe] Consider instrumenting the relevant libraries...
[omnitrace][exe]
[omnitrace][exe]        /lib64/libgcc_s.so.1
[omnitrace][exe]        /lib64/libpthread.so.0
[omnitrace][exe]        /lib64/libm.so.6
[omnitrace][exe]        /lib64/librt.so.1
[omnitrace][exe]        /home/ssitaram/cp2k-hip/libs/install/openmpi/lib/libmpi.so.40
[omnitrace][exe]        /opt/rocm-5.4.3//lib/libroctx64.so.4
[omnitrace][exe]        /opt/rocm-5.4.3//lib/libroctracer64.so.4
[omnitrace][exe]        /opt/rocm-5.4.3/hip/lib/libamdhip64.so.5
[omnitrace][exe]        /lib64/libstdc++.so.6
[omnitrace][exe]        /lib64/libc.so.6
[omnitrace][exe]        /lib64/ld-linux-x86-64.so.2
```

Path to new instrumented binary

Run Instrumented Binary

Binary Rewrite

```
$ omnitrace-instrument [omnitrace-options] -o <new-name-of-exec> -- <CMD> <ARGS>
```

Generating a new executable/library with instrumentation built-in:

```
$ omnitrace-instrument -o Jacobi_hip.inst -- ./Jacobi_hip
```

Run the instrumented binary:

```
$ mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```

Subroutine Instrumentation

Default instrumentation is main function and functions of 1024 instructions and more (for CPU)

To instrument routines with 500 or more cycles, add option "`-i 500`" (more overhead)

Binary rewrite is recommended for runs with multiple ranks as Omnitrace produces separate output files for each rank

```
[omnitrace][3624331][omnitrace_init_tooling] Instrumentation mode: Trace

   omnitrace v1.8.0
[953.765]       perfetto.cc:58656 Configured tracing session 1, #sources:1, duration:0 ms, #buffers:1, total buffer size:1024000 KB, total sessions:1, uid:0 session name: ""
Topology size: 1 x 1
Local domain size (current node): 4096 x 4096
[omnitrace][0][pid=3624331] MPI rank: 0 (0), MPI size: 1 (1)
Global domain size (all nodes): 4096 x 4096
Rank 0 selecting device 0 on host TheraC60
Starting Jacobi run.
Iteration:   0 - Residual: 0.022108
Iteration: 100 - Residual: 0.000625
Iteration: 200 - Residual: 0.000371
Iteration: 300 - Residual: 0.000274
Iteration: 400 - Residual: 0.000221
Iteration: 500 - Residual: 0.000187
Iteration: 600 - Residual: 0.000163
Iteration: 700 - Residual: 0.000145
Iteration: 800 - Residual: 0.000131
Iteration: 900 - Residual: 0.000120
Iteration: 1000 - Residual: 0.000111
Stopped after 1000 iterations with residue 0.000111
Total Jacobi run time: 1.5470 sec.
Measured lattice updates: 10.84 GLU/s (total), 10.84 GLU/s (per process)
Measured FLOPS: 184.36 GFLOPS (total), 184.36 GFLOPS (per process)
Measured device bandwidth: 1.04 TB/s (total), 1.04 TB/s (per process)

[omnitrace][3624331][0][omnitrace_finalize] finalizing...
[omnitrace][3624331][0][omnitrace_finalize]
[omnitrace][3624331][0][omnitrace_finalize] omnitrace/process/3624331 : 2.364423 sec wall_clock,  645.964 MB peak_rss,  388.739 MB page_rss, 4.330000 sec cpu_clock,  183.1 % cpu_util [laps: 1]
[omnitrace][3624331][0][omnitrace_finalize] omnitrace/process/3624331/thread/0 : 2.355893 sec wall_clock, 1.293230 sec thread_cpu_clock,   54.9 % thread_cpu_util,  645.964 MB peak_rss [laps: 1]
[omnitrace][3624331][0][omnitrace_finalize] omnitrace/process/3624331/thread/1 : 2.345084 sec wall_clock, 0.000261 sec thread_cpu_clock,    0.0 % thread_cpu_util,  642.676 MB peak_rss [laps: 1]
[omnitrace][3624331][0][omnitrace_finalize]
[omnitrace][3624331][0][omnitrace_finalize] Finalizing perfetto...
```

Generates traces for application run

Kernel Durations

```
$ cat omnitrace-Jacobi_hip.inst-output/2024-01-01_13.57/wall_clock-0.txt
```

If you do not see a `wall_clock.txt` dumped by Omnitrace, try modify the config file `$HOME/.omnitrace.cfg` and enable `OMNITRACE_PROFILE` (or prepend to your `mpirun` command):

```
…
OMNITRACE_PROFILE                                   = true
…
```

Durations

Call Stack

	0>>>	_MPI_Allreduce	1	5	wall_clock	sec	0.000012	0.000012	0.000012	0.000012	0.000000	0.000000	100.0
	0>>>	_hipDeviceSynchronize	1	5	wall_clock	sec	0.000019	0.000019	0.000019	0.000019	0.000000	0.000000	94.4
	0>>>	_NormKernel1(int, double, double, double const*, double*)	1	6	wall_clock	sec	0.000001	0.000001	0.000001	0.000001	0.000000	0.000000	100.0
	0>>>	_NormKernel2(int, double const*, double*)	1	6	wall_clock	sec	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	100.0
	0>>>	_MPI_Barrier	1	5	wall_clock	sec	0.000001	0.000001	0.000001	0.000001	0.000000	0.000000	100.0
	0>>>	_hipEventRecord	2	5	wall_clock	sec	0.000027	0.000014	0.000011	0.000016	0.000000	0.000003	100.0
	0>>>	_Halo D2H::Halo Exchange	1	5	wall_clock	sec	1.628420	1.628420	1.628420	1.628420	0.000000	0.000000	0.0
	0>>>	_hipStreamSynchronize	1	6	wall_clock	sec	0.000003	0.000003	0.000003	0.000003	0.000000	0.000000	100.0
	0>>>	_MPI Exchange::Halo Exchange	1	6	wall_clock	sec	1.628395	1.628395	1.628395	1.628395	0.000000	0.000000	0.0
	0>>>	_MPI_Waitall	1	7	wall_clock	sec	0.000002	0.000002	0.000002	0.000002	0.000000	0.000000	100.0
	0>>>	_Halo H2D::Halo Exchange	1	7	wall_clock	sec	1.628104	1.628104	1.628104	1.628104	0.000000	0.000000	0.0
	0>>>	_hipStreamSynchronize	1	8	wall_clock	sec	0.000003	0.000003	0.000003	0.000003	0.000000	0.000000	100.0
	0>>>	_hipLaunchKernel	5	8	wall_clock	sec	0.000615	0.000123	0.000005	0.000578	0.000000	0.000254	99.6
	0>>>	_mbind	1	9	wall_clock	sec	0.000003	0.000003	0.000003	0.000003	0.000000	0.000000	100.0
	0>>>	_hipMemcpy	1	8	wall_clock	sec	0.001122	0.001122	0.001122	0.001122	0.000000	0.000000	99.9
	0>>>	_LocalLaplacianKernel(int, int, int, double, double, double const*, double*)	1	9	wall_clock	sec	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	100.0
	0>>>	_HaloLaplacianKernel(int, int, int, double, double, double const*, double const*, double*)	1	9	wall_clock	sec	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	100.0
	0>>>	_JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*)	1	9	wall_clock	sec	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	100.0

Kernel Durations – Flat Profile

Edit in your `omnitrace.cfg` (or prepend to your `mpirun` command):

```
OMNITRACE_PROFILE                                = true
OMNITRACE_FLAT_PROFILE                           = true
```

Use flat profile to see aggregate duration of kernels and functions

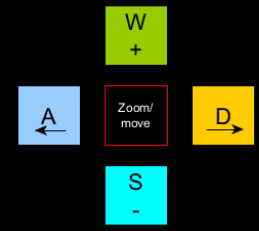
REAL-CLOCK TIMER (I.E. WALL-CLOCK TIMER)

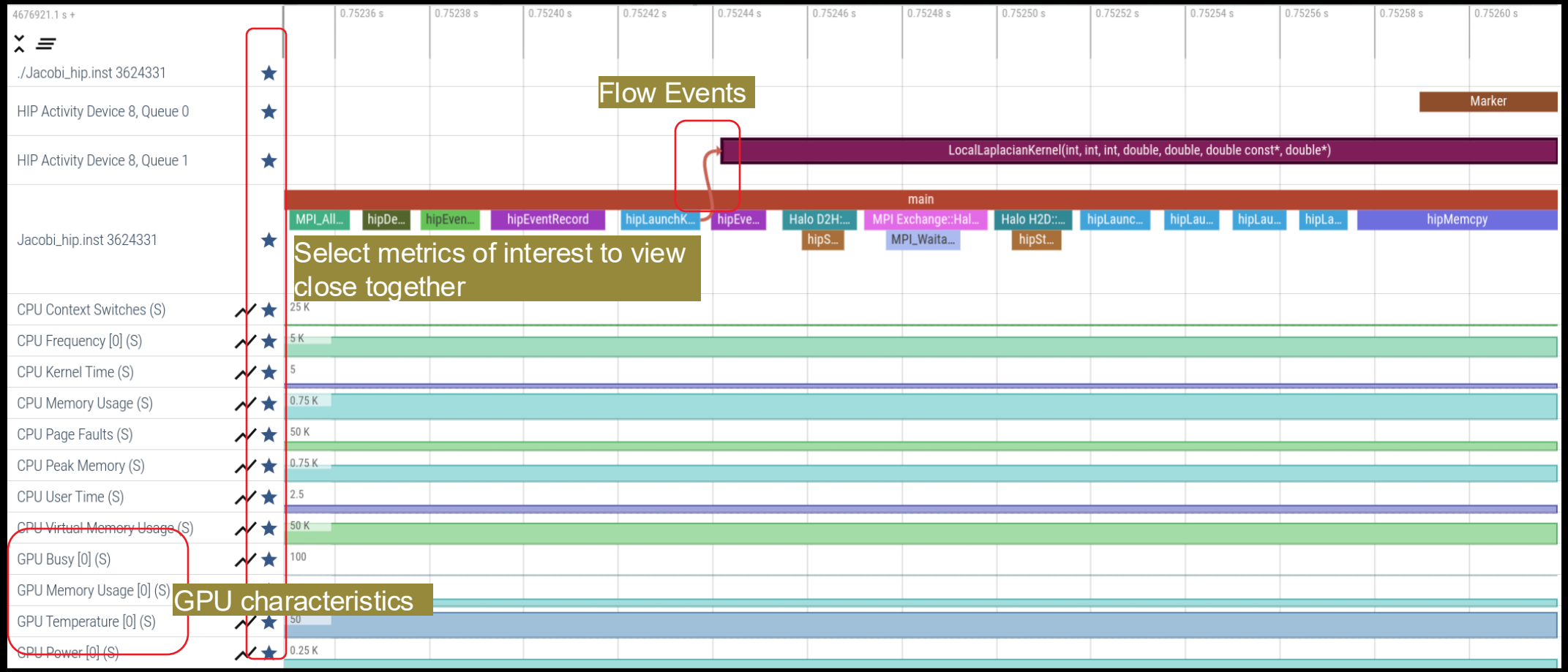
LABEL	COUNT	DEPTH	METRIC	UNITS	SUM	MEAN	MIN	MAX	VAR	STDDEV	% SELF
|0>>> main	1	0	wall_clock	sec	82.739099	82.739099	82.739099	82.739099	0.000000	0.000000	100.0
|0>>> MPI_Init	1	0	wall_clock	sec	34.056610	34.056610	34.056610	34.056610	0.000000	0.000000	100.0
|0>>> pthread_create	3	0	wall_clock	sec	0.014644	0.004881	0.001169	0.011974	0.000038	0.006145	100.0
|0>>> mbind	285	0	wall_clock	sec	0.001793	0.000006	0.000005	0.000020	0.000000	0.000002	100.0
|0>>> MPI_Comm_dup	1	0	wall_clock	sec	0.000212	0.000212	0.000212	0.000212	0.000000	0.000000	100.0
|0>>> MPI_Comm_rank	1	0	wall_clock	sec	0.000041	0.000041	0.000041	0.000041	0.000000	0.000000	100.0
|0>>> MPI_Comm_size	1	0	wall_clock	sec	0.000004	0.000004	0.000004	0.000004	0.000000	0.000000	100.0
|0>>> hipInit	1	0	wall_clock	sec	0.000372	0.000372	0.000372	0.000372	0.000000	0.000000	100.0
|0>>> hipGetDeviceCount	1	0	wall_clock	sec	0.000017	0.000017	0.000017	0.000017	0.000000	0.000000	100.0
|0>>> MPI_Allgather	1	0	wall_clock	sec	0.000009	0.000009	0.000009	0.000009	0.000000	0.000000	100.0
|0>>> hipSetDevice	1	0	wall_clock	sec	0.000024	0.000024	0.000024	0.000024	0.000000	0.000000	100.0
|0>>> hipHostMalloc	3	0	wall_clock	sec	0.126827	0.042276	0.000176	0.126453	0.005314	0.072900	100.0
|0>>> hipMalloc	7	0	wall_clock	sec	0.000458	0.000065	0.000024	0.000178	0.000000	0.000052	100.0
|0>>> hipMemset	1	0	wall_clock	sec	35.770403	35.770403	35.770403	35.770403	0.000000	0.000000	100.0
|0>>> hipStreamCreate	2	0	wall_clock	sec	0.016750	0.008375	0.005339	0.011412	0.000018	0.004295	100.0
|0>>> hipMemcpy	1005	0	wall_clock	sec	8.506781	0.008464	0.000610	0.039390	0.000023	0.004844	100.0
|0>>> hipEventCreate	2	0	wall_clock	sec	0.000037	0.000018	0.000016	0.000021	0.000000	0.000003	100.0
|0>>> hipLaunchKernel	5002	0	wall_clock	sec	0.181301	0.000036	0.000025	0.012046	0.000000	0.000278	100.0
|0>>> MPI_Allreduce	1003	0	wall_clock	sec	0.002009	0.000002	0.000001	0.000022	0.000000	0.000001	100.0
|0>>> hipDeviceSynchronize	1001	0	wall_clock	sec	0.016813	0.000017	0.000015	0.000043	0.000000	0.000004	100.0
|0>>> MPI_Barrier	3	0	wall_clock	sec	0.000007	0.000002	0.000001	0.000004	0.000000	0.000001	100.0
|0>>> hipEventRecord	2000	0	wall_clock	sec	0.046701	0.000023	0.000020	0.000225	0.000000	0.000006	100.0
|0>>> hipStreamSynchronize	2000	0	wall_clock	sec	0.030366	0.000015	0.000013	0.000382	0.000000	0.000009	100.0
|0>>> MPI_Waitall	1000	0	wall_clock	sec	0.001665	0.000002	0.000002	0.000007	0.000000	0.000000	100.0
|0>>> NormKernel1(int, double, double, double const*, double*)	1001	0	wall_clock	sec	0.001502	0.000002	0.000001	0.000006	0.000000	0.000000	100.0
|0>>> NormKernel2(int, double const*, double*)	1000	0	wall_clock	sec	0.001972	0.000002	0.000001	0.000003	0.000000	0.000001	100.0
|0>>> LocalLaplacianKernel(int, int, int, double, double, double const*, double*)	1000	0	wall_clock	sec	0.001488	0.000001	0.000001	0.000007	0.000000	0.000000	100.0
|0>>> HaloLaplacianKernel(int, int, int, double, double, double const*, double const*, double*)	1000	0	wall_clock	sec	0.001465	0.000001	0.000001	0.000007	0.000000	0.000000	100.0
|0>>> hipEventElapsedTime	1000	0	wall_clock	sec	0.015060	0.000015	0.000014	0.000041	0.000000	0.000002	100.0
|0>>> JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*)	1000	0	wall_clock	sec	0.002598	0.000003	0.000001	0.000006	0.000000	0.000001	100.0
|0>>> pthread_join	1	0	wall_clock	sec	0.000396	0.000396	0.000396	0.000396	0.000000	0.000000	100.0
|0>>> hipFree	4	0	wall_clock	sec	0.000526	0.000131	0.000021	0.000243	0.000000	0.000091	100.0
|0>>> hipHostFree	2	0	wall_clock	sec	0.000637	0.000318	0.000287	0.000350	0.000000	0.000044	100.0
|3>>> start_thread	1	0	wall_clock	sec	0.004802	0.004802	0.004802	0.004802	0.000000	0.000000	100.0
|1>>> start_thread	1	0	wall_clock	sec	81.987779	81.987779	81.987779	81.987779	0.000000	0.000000	100.0
|2>>> start_thread	-	0	-	-	-	-	-	-	-	-	-

Visualizing Trace (3/3)

Use Perfetto

Zoom in to investigate regions of interest





Configure Omnitrace to Collect GPU Hardware Counters

Modify config file

Modify the config file `$HOME/.omnitrace.cfg` to add desired metrics and for concerned `GPU#ID`:

```
…
OMNITRACE_ROCM_EVENTS = FetchSize:device=0, VALUUtilization:device=0, MemUnitBusy:device=0
…
```

To profile desired metrics for all participating GPUs:

```
…
OMNITRACE_ROCM_EVENTS = FetchSize, VALUUtilization, MemUnitBusy
…
```

Note: currently experiencing issues with ROCm 6.2.1, fix coming soon

Full list of GPU metrics at https://github.com/ROCm/rocprofiler/blob/amd-staging/test/tool/metrics.xml

Execution with Hardware Counters

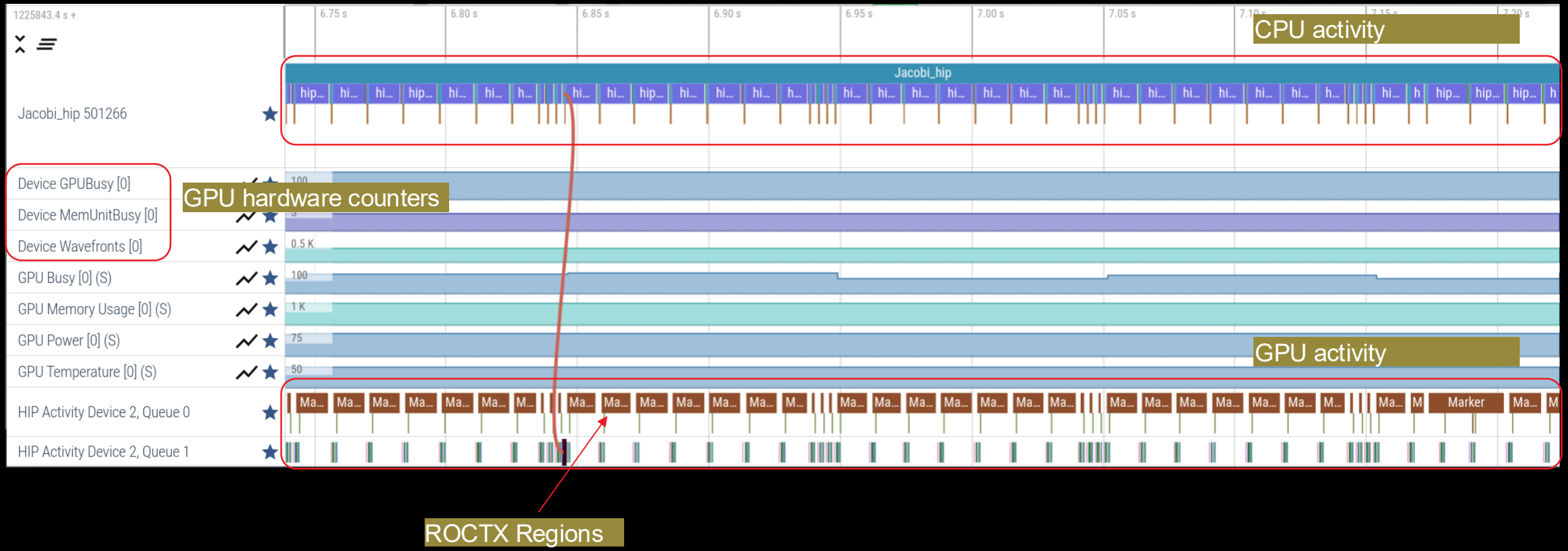
After modifying .cfg file to set up `OMNITRACE_ROCM_EVENTS` with GPU metrics run:

```
$ mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```

```
[omnitrace][1056814][0][omnitrace_finalize]
[omnitrace][1056814][0][omnitrace_finalize] Finalizing perfetto...
[omnitrace][1056814][perfetto]> Outputting '/datasets/teams/dcgpu_training/lstanisi/test_hackmd3/HPCTrainingExamples/HIP/jacobi/omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/perfetto-trace-0.proto' (8130.87 KB / 8.13 MB / 0.01 GB)... Done
[omnitrace][1056814][rocprof-device-0-FetchSize]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-FetchSize-0.json'
[omnitrace][1056814][rocprof-device-0-FetchSize]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-FetchSize-0.txt'
[omnitrace][1056814][rocprof-device-0-VALUUtilization]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-VALUUtilization-0.json'
[omnitrace][1056814][rocprof-device-0-VALUUtilization]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-VALUUtilization-0.txt'
[omnitrace][1056814][rocprof-device-0-MemUnitBusy]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-MemUnitBusy-0.json'
[omnitrace][1056814][rocprof-device-0-MemUnitBusy]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/rocprof-device-0-MemUnitBusy-0.txt'
[omnitrace][1056814][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/wall_clock-0.json'
[omnitrace][1056814][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/wall_clock-0.txt'
[omnitrace][1056814][roctracer]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/roctracer-0.json'
[omnitrace][1056814][roctracer]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/roctracer-0.txt'
[omnitrace][1056814][metadata]> Outputting 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/metadata-0.json' and 'omnitrace-Jacobi_hip.inst-output/2024-10-02_10.36/functions-0.json'
```

GPU hardware counters

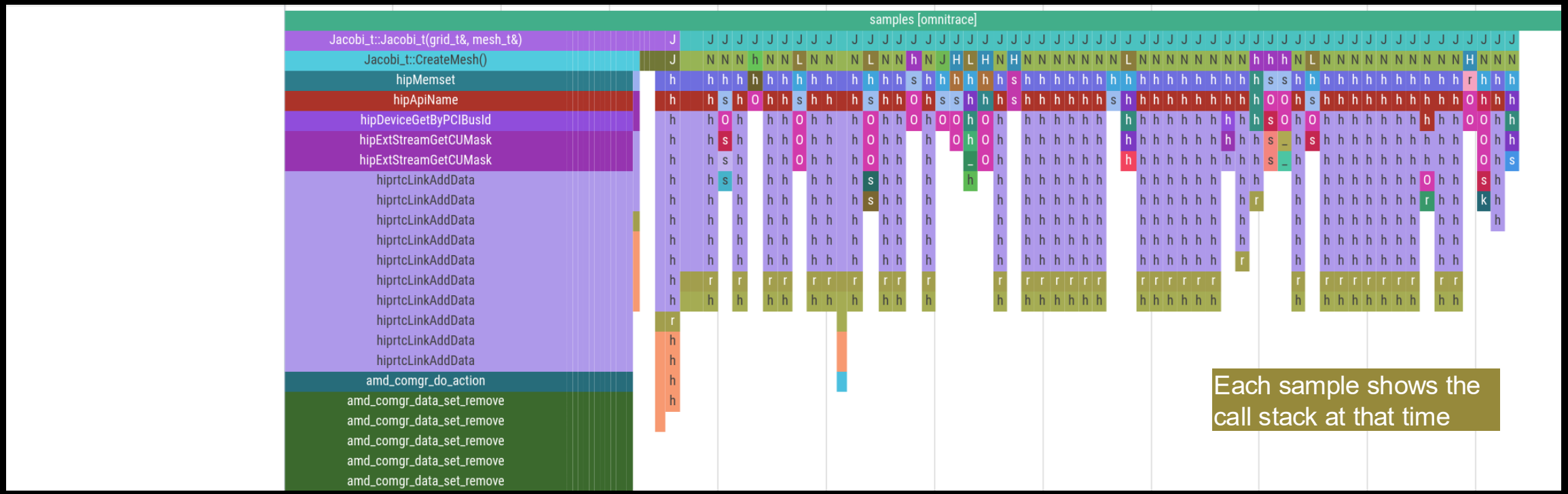
Visualization with Hardware Counters



Sampling CPU Call-Stack (1/2)

`OMNITRACE_USE_SAMPLING = true; OMNITRACE_SAMPLING_FREQ = 100` (100 samples per second)

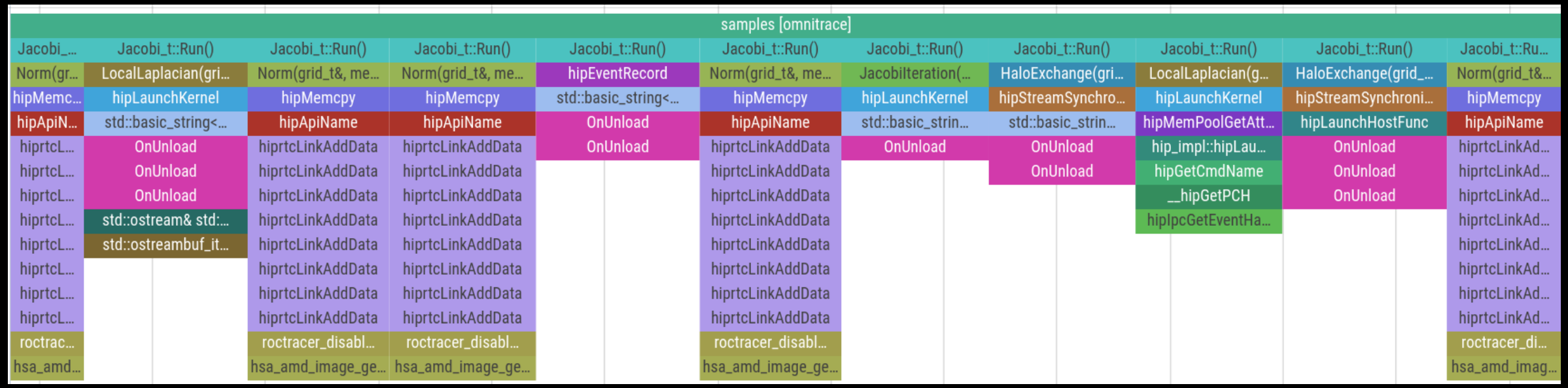
Alternatively run with `omnitrace-sample`



Scroll down all the way in Perfetto to see the sampling output

Sampling CPU Call-Stack (2/2)

Zoom in call-stack sampling



Thread 0 (S) 3625610 ← Sampling data is annotated with (S)

Additional Features

- Dynamic runtime instrumentation
- User API to control instrumentation
- `OMNITRACE_USE_KOKKOSP=true` supports Kokkos profiling
- `omnitrace-python` supports Python™ profiling (only with AMD Research ROCm)
- `omnitrace-causal` for invoking causal profiling (experimental)

Fixes coming soon:

- Hardware counters
- Full OpenMP® support
- Visualizing traces from multiple MPI ranks

Summary

- Omnitrace - powerful tool to understand CPU + GPU activity on AMD GPUs
  - Ideal for an initial look at how an application runs
  - Easy to visualize traces in Perfetto

- Leverages several other tools and combines their data into a comprehensive output files
  - Some tools used are AMDμProf, rocprofiler, rocm-smi, roctracer, perf, etc.

- **Helps users analyze overlaps between CPU/GPU compute and communication**

Other profiling options

Presenter: Sam Antao

LUMI Pre-hackathon training

October 8th , 2024

AMD

together we advance_

Hands-on Exercises

https://hackmd.io/@sfantao/lumi-prehack-may-2025

We encourage you to look at our HPC Training Examples repo for other examples:
https://github.com/amd/HPCTrainingExamples

A table of contents for the READMEs if available at the top-level README in the repo

Rocprofv3 exercises instructions: Rocprofv3/README.md
Link to instructions on how to run Omnitrace tests: Omnitrace/omnitrace_jacobi/MI200/README.md

Questions?

ssh <you user>@lumi.csc.fi

https://hackmd.io/@sfantao/lumi-prehack-may-2025

DISCLAIMERS AND ATTRIBUTIONS


AMD