Introduction to ROC-Profiler (`rocprofv3`)

Gina Sitaraman, George Markomanolis, Justin Chang, Julio Maia, Noel Chalmers, Paul T. Bauman, Nicholas Curtis, Nicholas Malaya, Alessandro Fanfarillo, Jose Noudohouenou, Chip Freitag, Damon McDougall, Noah Wolfe, Jakub Kurzak, Samuel Antao, Bob Robey

Tutorial at CRAY USER GROUP
May 5, 2025

AMD
together we advance_

Logistics

- Registration for exercises:
  - First Name
  - Last Name
  - Email
  - Country
- Email: georgios.markomanolis@amd.com

- Access:
  - ssh $USER@aac6.amd.com -p 7001

- Exercises URL:
  - https://hackmd.io/@gmarkoma/rocprofv3_cug2025
  - https://hackmd.io/@gmarkoma/cug2025-AMDGPUProfiling#Rocprofiler-Systems-Rocprofsys
  - https://hackmd.io/@gmarkoma/cug2025-AMDGPUProfiling#Rocprof-compute

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 vs rocprofv2 vs rocprofv3: What is the difference?

	rocprof	rocprofv2	rocprofv3
Brief description	Legacy tool for tracing and performance counter collection	Additional functionalities, AMDGPUs support and output formats	More efficient and flexible tool with advanced features + emphasis on stability and robustness
Underlying libraries	rocprofiler, roctracer		rocprofiler_sdk
Output formats	CSV, JSON	CSV, JSON, Pftrace	CSV, JSON, Pftrace, OTF2
Visualization format	JSON (Perfetto)	Pftrace (Perfetto)	Pftrace (Perfetto), OTF2 (Vampir)
ROCm docs pages	RocProfiler and RocTracer		rocprofiler_sdk
Status	Only critical bug fixes	Not maintained anymore	**Beta, under active development**

Detailed comparison of profiling tools: https://github.com/ROCm/rocprofiler-sdk/blob/amd-mainline/source/docs/conceptual/comparing-with-legacy-tools.rst

Background – AMD Profilers

ROC-profiler (rocprofv3)

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

	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

Rocprof-sys

Trace collection
- Comprehensive trace collection
- CPU
- GPU

Supports
- CPU copy
- HIP API
- HSA API
- GPU Kernels
- OpenMP ®
- MPI
- Kokkos
- p-threads
- multi-GPU

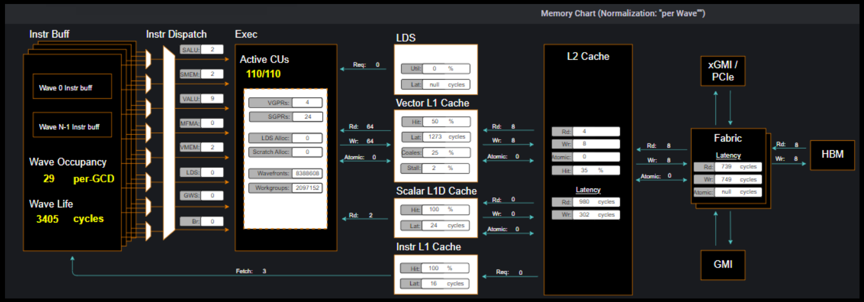
Visualisation
- Traces visualized with Perfetto

Rocprof-compute

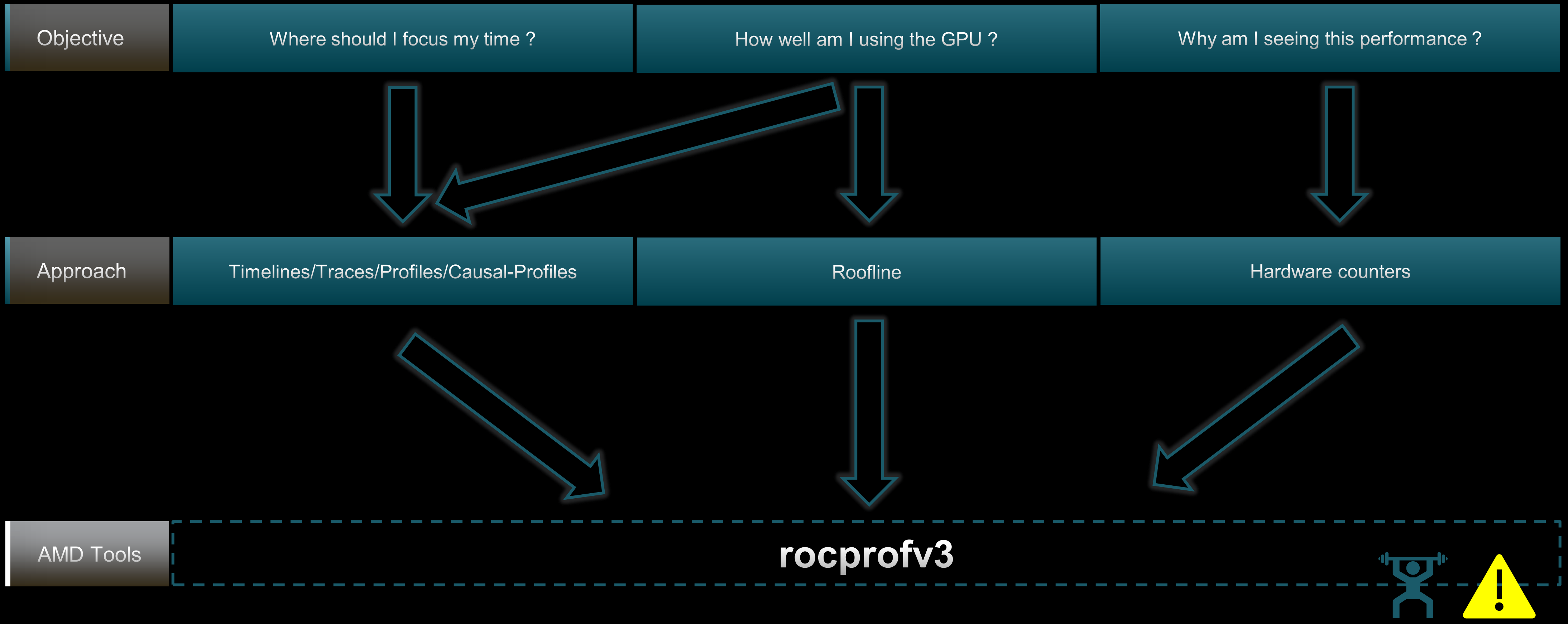
Performance Analysis
- Automated collection of hardware counters
- Analysis
- Visualisation

Supports
- Speed of Light
- Memory chart
- Rooflines
- Kernel comparison

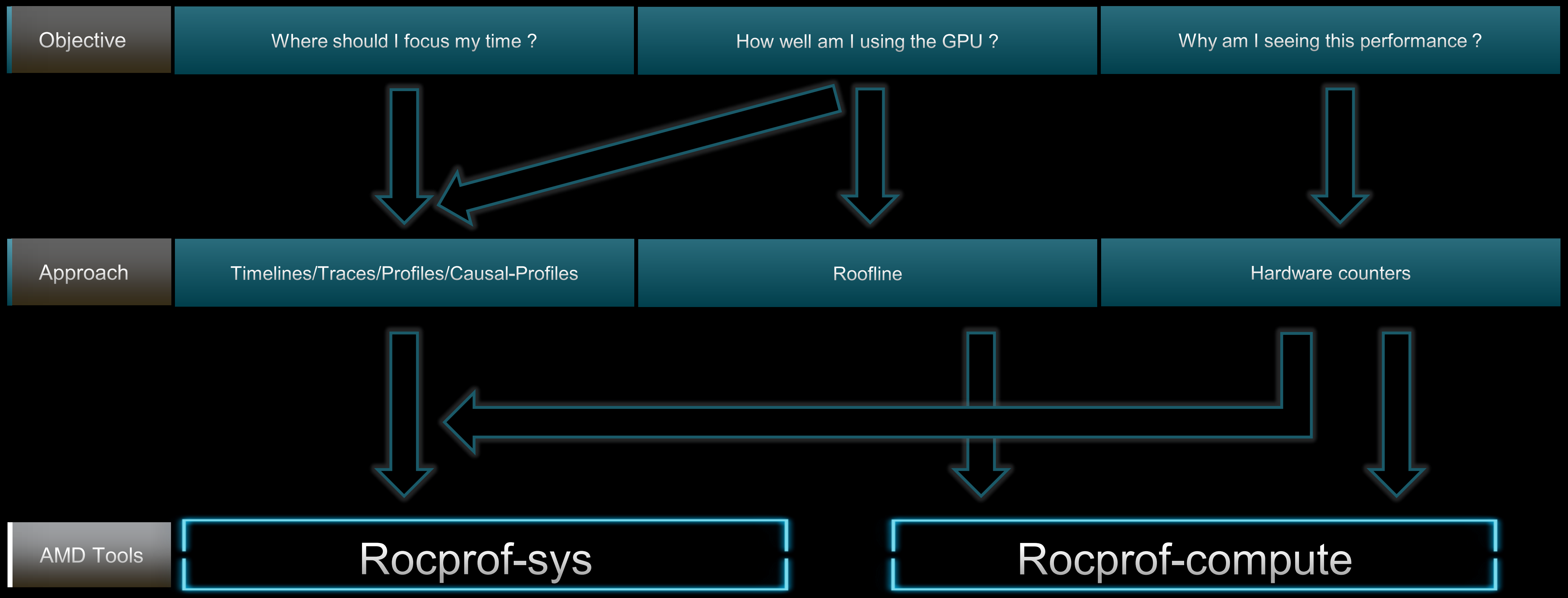
Visualisation
- With Grafana or standalone GUI



Background – AMD Profilers



Background – AMD Profilers



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)
  - `-s, --sys-trace` For Collecting HIP, HSA, Marker (ROCTx), Memory copy, Scratch memory, and Kernel dispatch traces
  - `-i INPUT, --input INPUT`
  - Input file for counter collection
  - `--kernel-names KERNEL_NAMES [KERNEL_NAMES ...]`
  - Filter kernel names

rocprofv3: Getting Started + Useful Flags (II)

- Useful housekeeping flags:
  - -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
  - -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
  - --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

- rocprofv3 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`

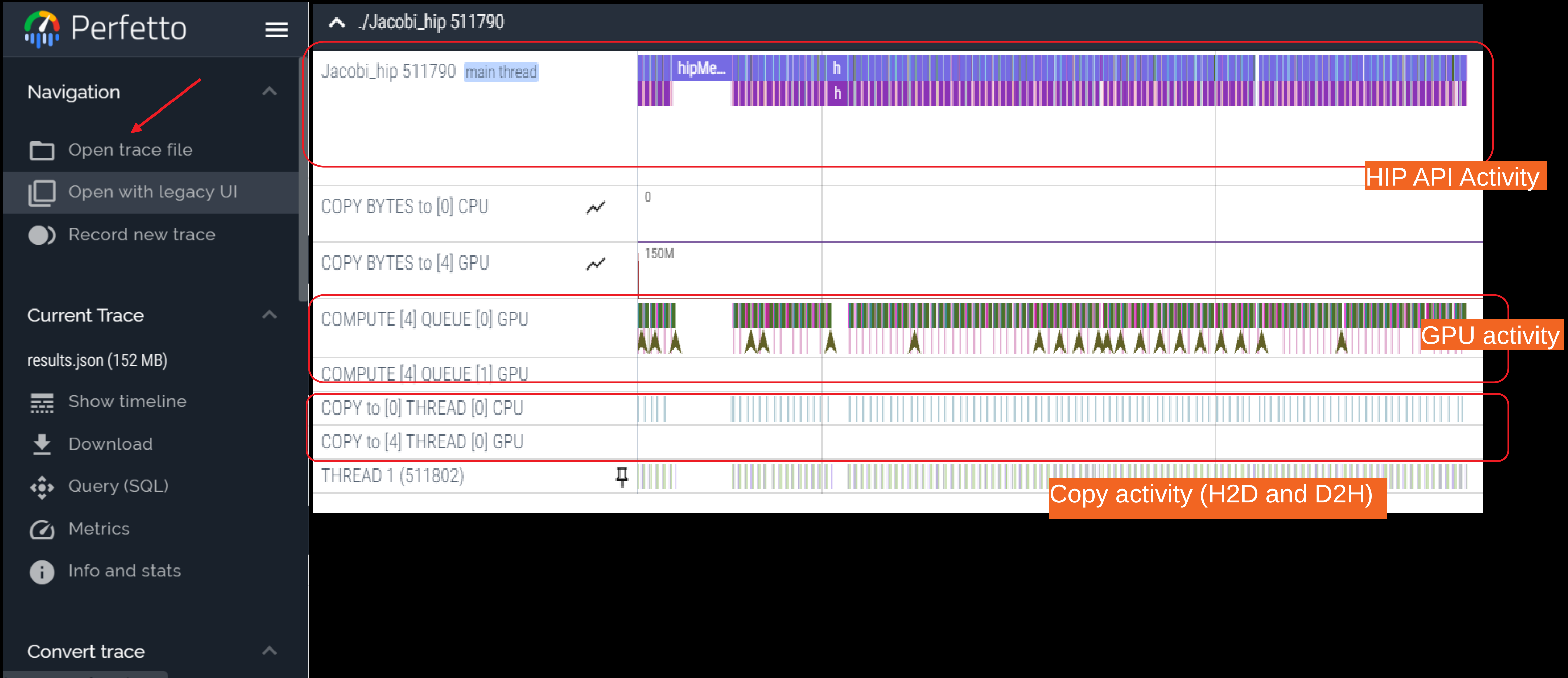
- You can combine modes like --stats `--hip-trace --hsa-trace --output-format pftrace`
- `Pftrace file output format is more stable to visualize with Perfetto (--output-format pftrace)`

rocprofv3 + Perfetto: Collecting and Visualizing Application Traces

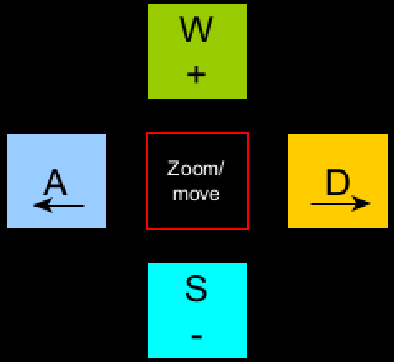
- rocprofv3 can collect traces

```
$ /opt/rocm/bin/rocprofv3 --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/)



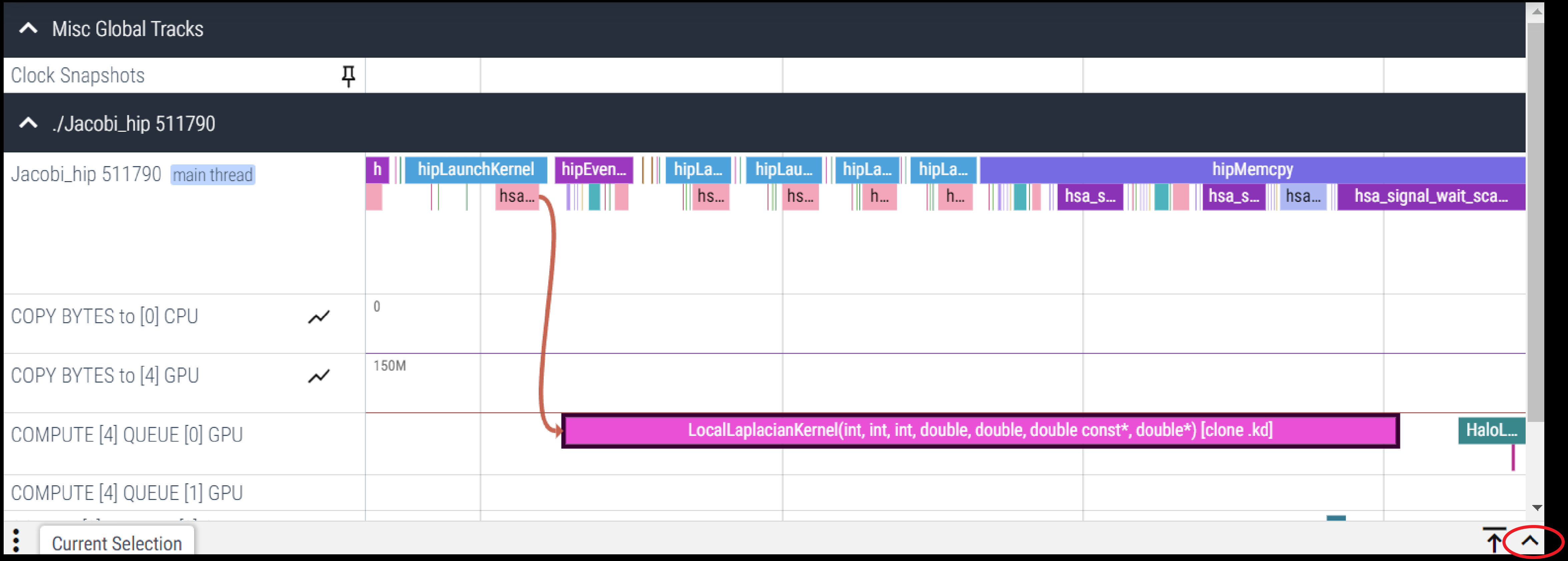
Perfetto: Visualizing Application Traces



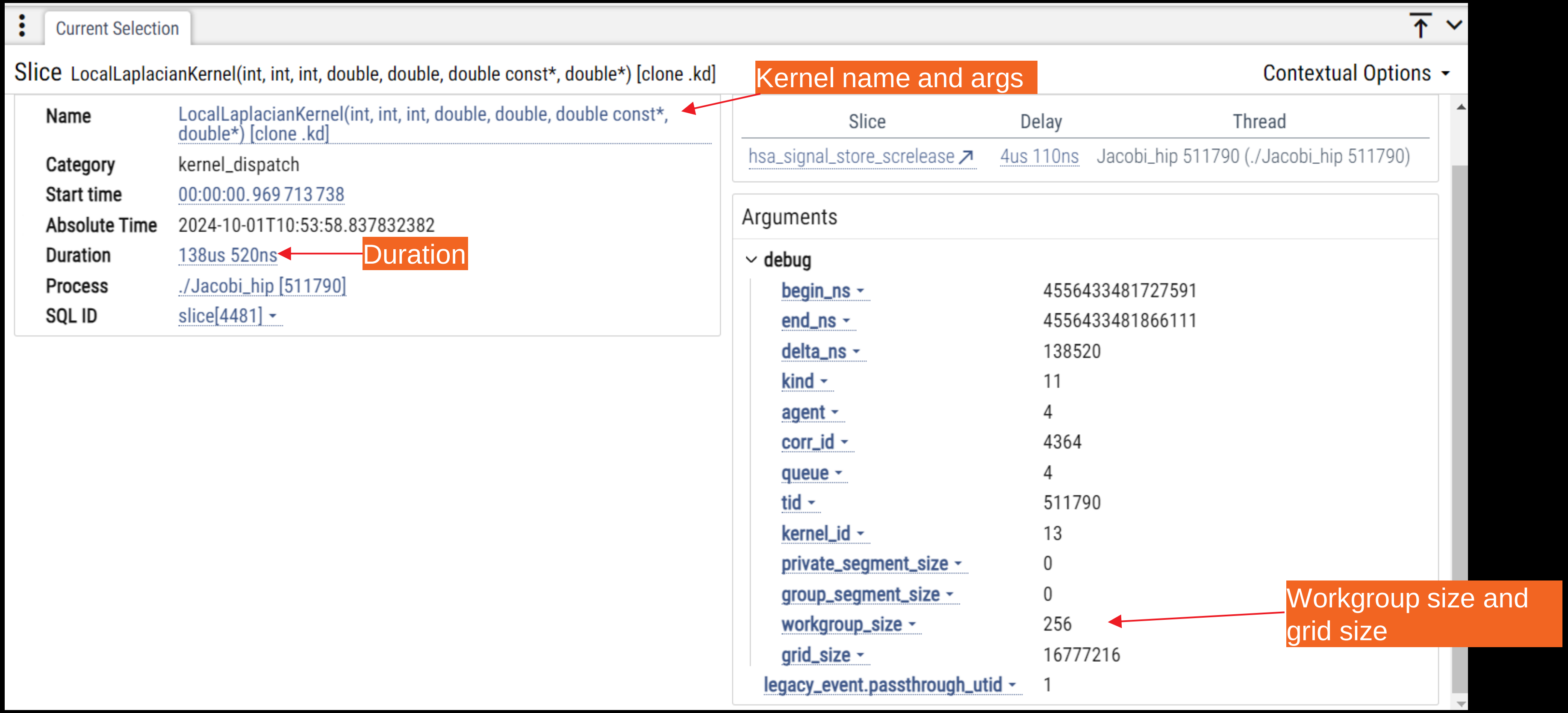
- Zoom in to see individual events
- Navigate trace using WASD keys

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:

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

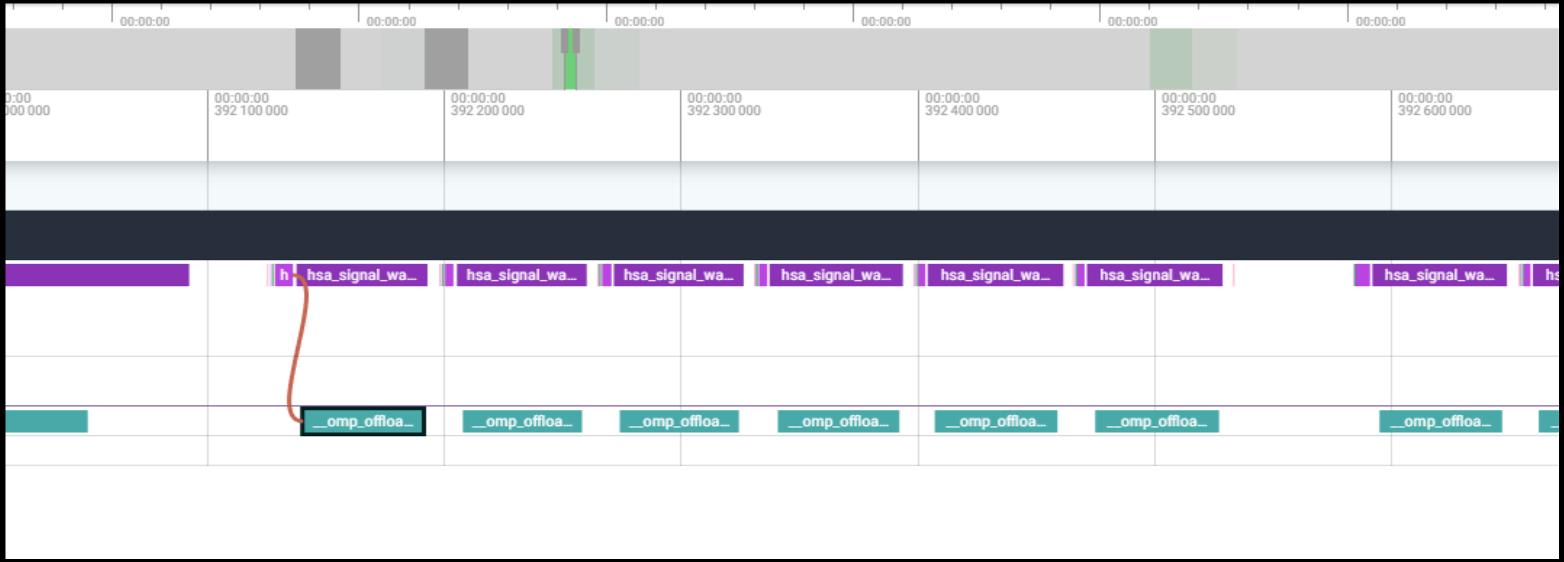
Content of XXXXX_kernel_stats.csv:

```
"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
```

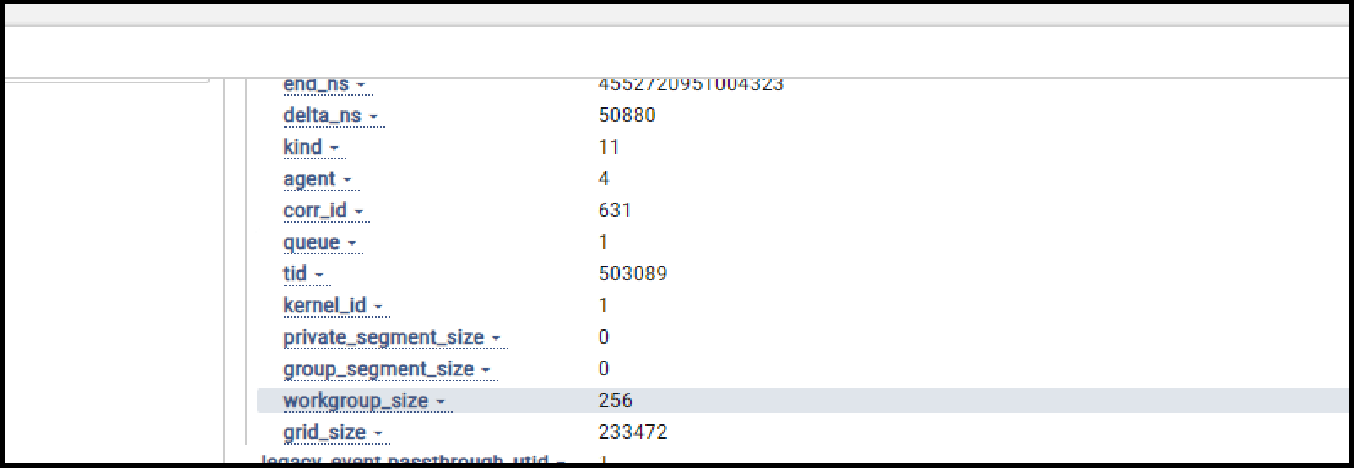
Content of XXXXX_kernel_trace.csv

```
"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



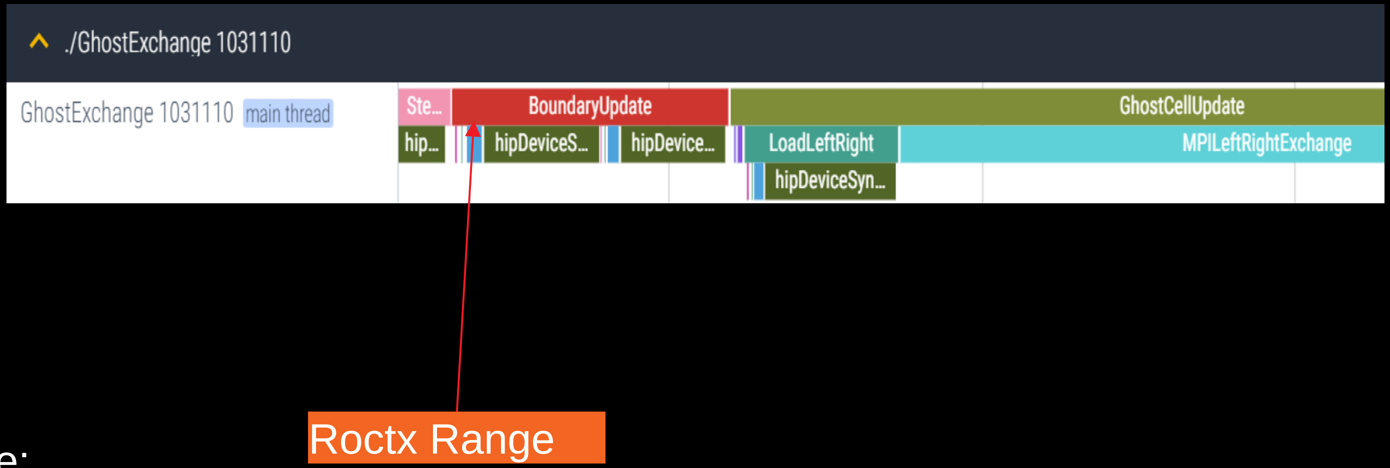
- Using: *--sys-trace --output-format pftrace*
- We can use: *--kernel-trace --output-format pftrace*



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

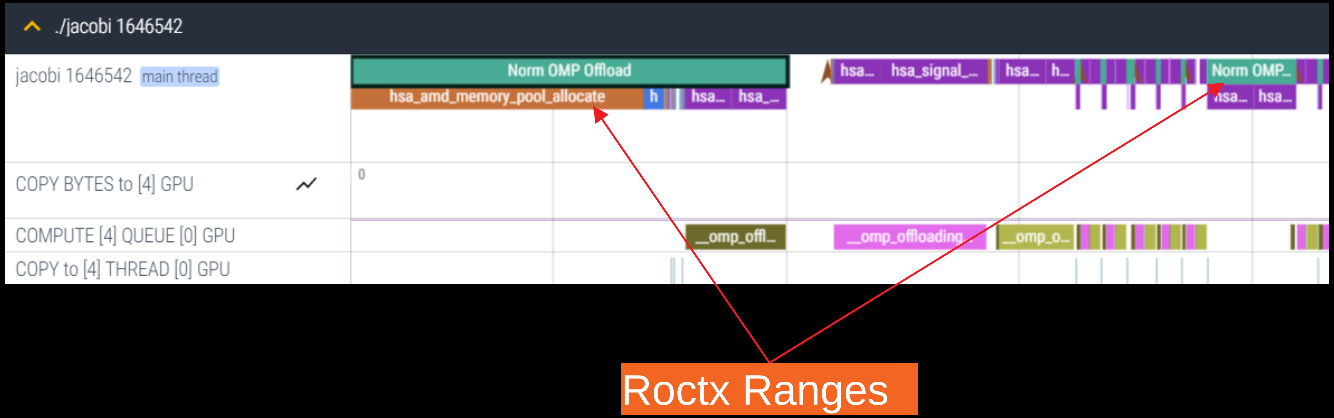
Collecting Application Traces with roctx Regions and Markers

- `rocprofv3` can collect user defined regions or markers using `roctx`
  1. Include `hip_profiling.F` (see next slide for details) into declaration

```
…
use hip_profiling, only: roctxRangePushA,&
                         roctxRangePop,&
                         roctxMarkA
use iso_c_binding, only: c_null_char
…
```

  2. Annotate code with `roctx` regions or marker:

```
…
ret = roctxRangePushA(“Range name"//c_null_char)
function()
call roctxRangePop()
...
call roctxMarkA("Marker"//c_null_char)
…
! Note: Fortran strings are represented with a length descriptor and do not require a null terminator,
!       but C strings are represented as character arrays requiring a null terminator (`\0`)
```



  3. Adapt Makefile by adding `hip_profiling.o`, and extending link line with `roctx` library:

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

- Profile with `--marker-trace` or `--sys-trace` option:

```
$ rocprofv3 --kernel-trace --marker-trace --output-format pftrace -- <app with arguments>
```

- Be careful about differences between `rocprof` and `rocprofv3` regarding `roctx`

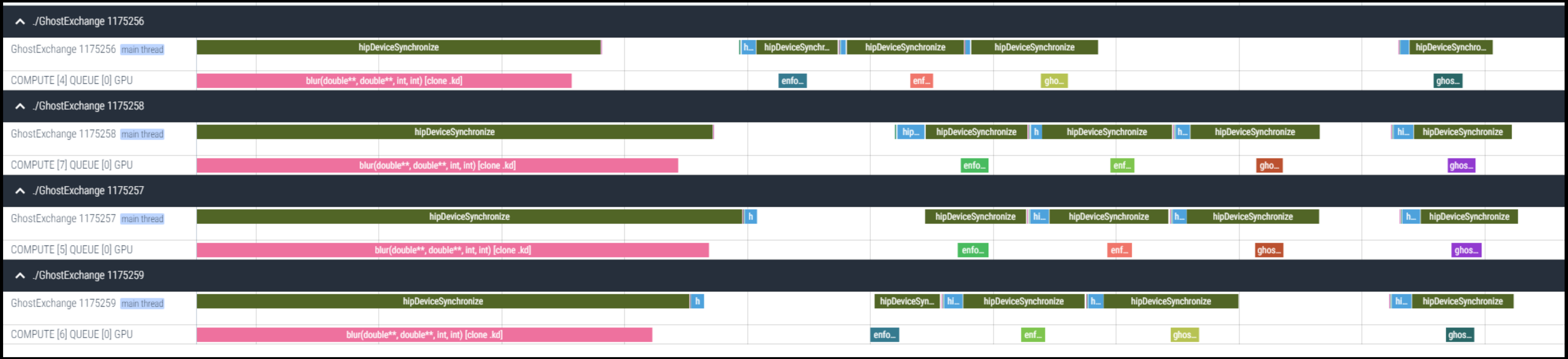
roctx Regions: hip_profiling.F

Coming soon in hipfort

```
MODULE hip_profiling

 INTERFACE
     SUBROUTINE roctxMarkA(message) BIND(c, name="roctxMarkA")
       USE ISO_C_BINDING,   ONLY: C_CHAR
       IMPLICIT NONE
       CHARACTER(C_CHAR) :: message(*)
     END SUBROUTINE roctxMarkA

     FUNCTION roctxRangePushA(message) BIND(c, name="roctxRangePushA")
       USE ISO_C_BINDING,   ONLY: C_INT,&
                                  C_CHAR
       IMPLICIT NONE
       INTEGER(C_INT) :: roctxRangePushA
       CHARACTER(C_CHAR) :: message(*)
     END FUNCTION roctxRangePushA

     SUBROUTINE roctxRangePop() BIND(c, name="roctxRangePop")
       IMPLICIT NONE
     END SUBROUTINE roctxRangePop

 END INTERFACE

END MODULE hip_profiling
```

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



together we advance_

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

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

Performance Counters Tips and Tricks

- GPU Hardware counters are global
  - Kernel dispatches are serialized to ensure that only one dispatch is ever in flight
  - It is recommended that no other applications are using the GPU when collecting performance counters

- Use `-T on` which will report only kernel names, leaving off kernel arguments

- How do you time a kernel's duration?
  - `$ /opt/rocm/bin/rocprofv3 --kernel-trace -- <app with args>`
  - This produces two times: *Start_Timestamp, End_Timestamp*
  - Closest thing to a kernel duration: *End_Timestamp* - *Start_Timestamp*
  - If you run with "`--stats`" the resultant XXXXX_kernel_stats.csv file will include a kernel duration column TotalDurationNs and one AverageNs
    - Note: the duration is aggregated over repeated calls to the same kernel

rocprofv3: 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

- rocprofv3 is the open source, command line AMD GPU profiling tool distributed with ROCm 6.2 and later
- rocprofv3 provides tracing of GPU kernels, through various options, HIP API, HSA API, Copy activity and others
- rocprofv3 can be used with MPI applications directly
- rocprofv3 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, json, OTF2 format

Questions?

Rocprof-sys: Performance Analysis Tools for AMD GPUs

Presenter: George Markomanolis

Contributors: Gina Sitaraman, George Markomanolis, Jonathan Madsen, Austin Ellis, Bob Robey, Xiaomin Lu, Noah Wolfe, Samuel Antao

Tutorial at CRAY USER GROUP
May 5, 2025

AMD
together we advance_

Logistics

- Registration for exercises:
  - First Name
  - Last Name
  - Email
  - Company
  - Company address
  - Country
  - State/Province
  - Phone
- Email: georgios.markomanolis@amd.com

- Access:
  - ssh $USER@aac6.amd.com -p 7001

- Exercises URL:
  - https://hackmd.io/@gmarkoma/cug2025-AMDGPUProfiling#Rocprofiler-Systems-Rocprofsys
  - https://hackmd.io/@gmarkoma/cug2025-AMDGPUProfiling#Rocprof-compute

Introduction to Rocprof-sys (ex-~~Omnitrace~~)

AMD
together we advance_

[Public]

Background – AMD Profilers

ROC-profiler (rocprofv3)

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

	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

Rocprof-sys

Trace collection
- Comprehensive trace collection
  - CPU
  - GPU

Supports
- CPU copy
- HIP API
- HSA API
- GPU Kernels
- OpenMP ®
- MPI
- Kokkos
- p-threads
- multi-GPU

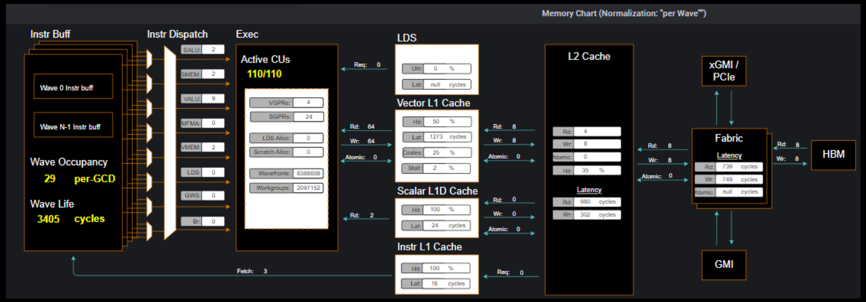
Visualisation
- Traces visualized with Perfetto

Rocprof-compute

Performance Analysis
- Automated collection of hardware counters
  - Analysis
  - Visualisation

Supports
- Speed of Light
- Memory chart
- Rooflines
- Kernel comparison

Visualisation
- With Grafana or standalone GUI

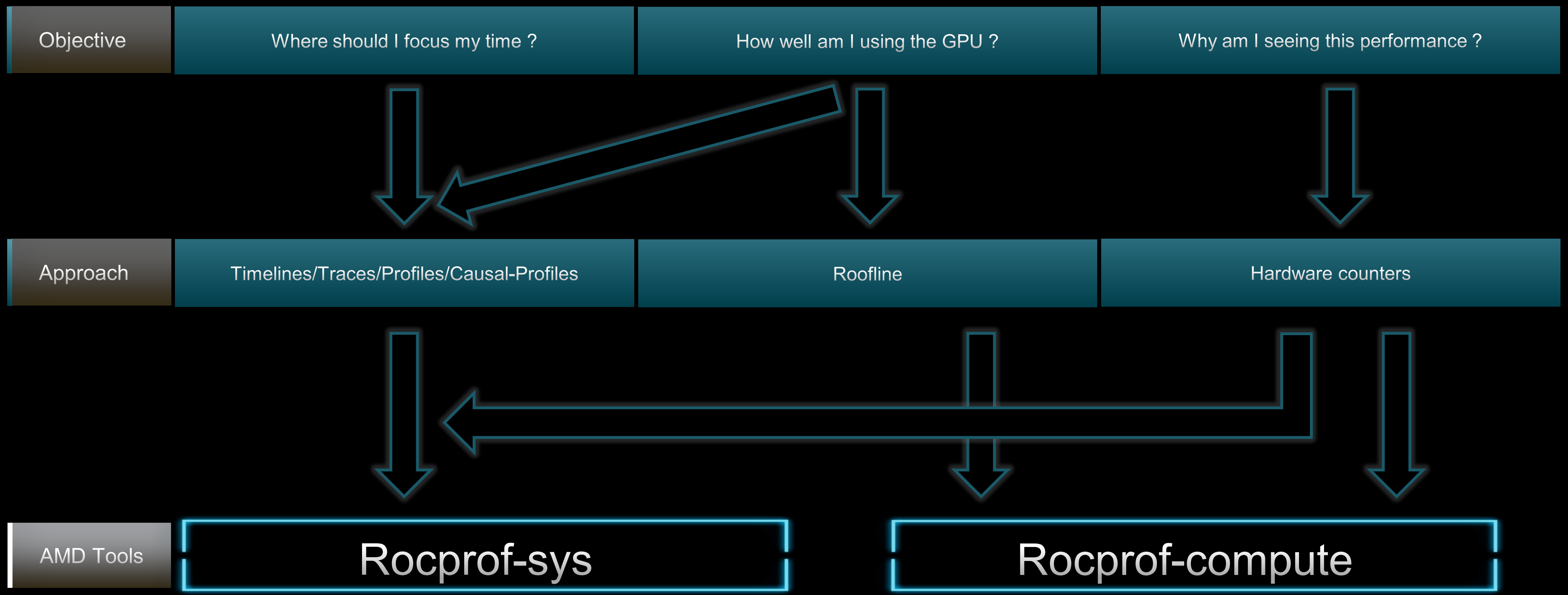


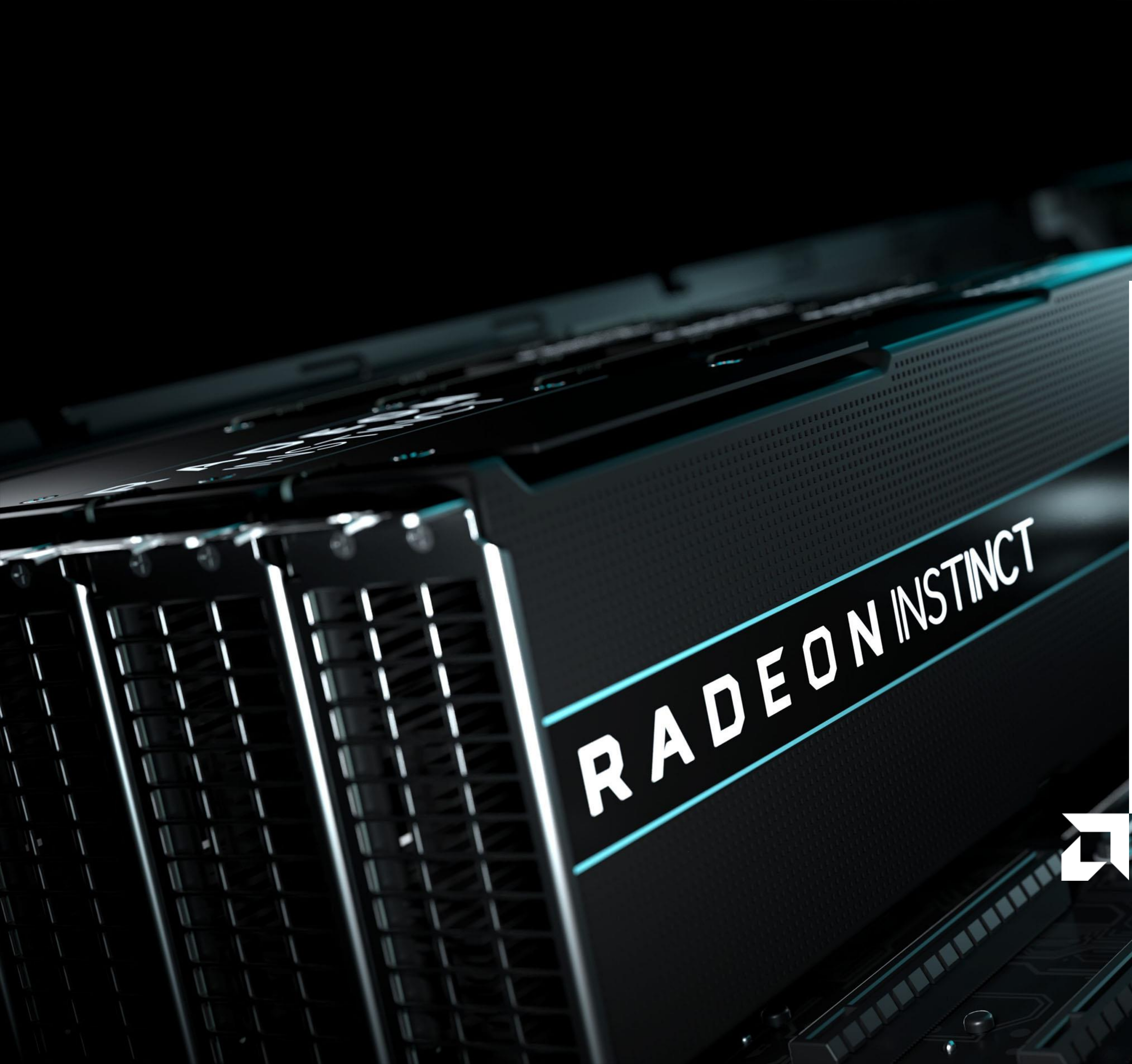
Background – AMD Profilers

Objective	Where should I focus my time ?	How well am I using the GPU ?	Why am I seeing this performance ?
Approach	Timelines/Traces/Profiles/Causal-Profiles	Roofline	Hardware counters
AMD Tools	rocprofv3		

[Public]

Background – AMD Profilers





Rocprof-sys

Rocprof-sys: Application Profiling, Tracing, and Analysis

- **AMD Product (ex-Research Tool)**
  - Repository: https://github.com/ROCm/rocprofiler-systems
  - Included in ROCm starting with 6.3.0
- **Language Support**
  - C/C++
  - Fortran
  - Python
  - OpenCL ™
- **Data Collection Modes**
  - Dynamic instrumentation
  - Statistical/process sampling
  - Causal Profiling
- **Data Analysis**
  - High-level summary
  - Comprehensive trace
  - Critical trace analysis
- **Parallelism Support**
  - MPI
  - OpenMP®
  - Pthreads
  - HIP
  - HSA
  - Kokkos
- **GPU Metrics**
  - HW counters
  - HSA API
  - HIP API
  - HIP trace
  - HSA trace
  - Memory & thermal
- **CPU Metrics**
  - HW counters
  - Timing metrics
  - Memory access
  - Network
  - I/O
  - more…

Refer to current documentation for recent updates

Installation (if required)

To use pre-built binaries, select the version that matches your operating system, ROCm version, etc.

Select OpenSuse operating system for HPE/AMD system:

rocprofiler-systems-1.0.0-opensuse-15.5-ROCm-60300-PAPI-OMPT-Python3.sh

There are .rpm and .deb files for installation also. In future versions, binary installers for RHEL also available.

Full documentation: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/index.html

```
wget https://github.com/ROCm/rocprofiler-systems/releases/latest/download/rocprofiler-systems-install.py
python3 ./rocprofiler-systems-install.py --prefix /opt/rocprofiler-systems --rocm 6.3
```

Note: If installing from source, remember to clone the rocprof-sys repo recursively

Rocprof-sys instrumentation Modes

Runtime Instrumentation	Dynamic binary instrumentation		
	Characterize performance	Sample every invocation	Large overheads
Sampling Instrumentation	Periodic sampling of entire application		
	Statistical sampling	Process sampling	

Basic command-line syntax:

```
$ rocprof-sys-run [rocprof-sys-options] -- <CMD> <ARGS>
```

For more information or help use -h/--help/? flags:

```
$ rocprof-sys-run -h
```

Can also execute on systems using a job scheduler. For example, with SLURM, an interactive session can be used as:

```
$ srun [options] rocprof-sys-run [rocprof-sys-options] -- <CMD> <ARGS>
```

Documentation: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/index.html

Rocprof-sys Configuration

```
$ rocprof-sys-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:

```
$ rocprof-sys-avail -h
```

Collect information for rocprof-sys-related settings using shorthand -c for --categories :

```
$ rocprof-sys-avail –c perfetto
```

```
$ rocprof-sys-avail -c perfetto
|-------------------------------------|-----------------|-------------------------------------------------------------------------|
|        ENVIRONMENT VARIABLE         |      VALUE      |                              CATEGORIES                                 |
|-------------------------------------|-----------------|-------------------------------------------------------------------------|
| ROCPROFSYS_PERFETTO_BACKEND         |    inprocess    | custom, librocprof-sys, perfetto, rocprofsys                            |
| ROCPROFSYS_PERFETTO_BUFFER_SIZE_KB  |     1024000     | custom, data, librocprof-sys, perfetto, rocprofsys                      |
| ROCPROFSYS_PERFETTO_FILL_POLICY     |     discard     | custom, data, librocprof-sys, perfetto, rocprofsys                      |
| ROCPROFSYS_TRACE                    |      true       | backend, custom, librocprof-sys, perfetto, rocprofsys                   |
| ROCPROFSYS_TRACE_DELAY              |        0        | custom, librocprof-sys, perfetto, profile, rocprofsys, timemory, trace  |
| ROCPROFSYS_TRACE_DURATION           |        0        | custom, librocprof-sys, perfetto, profile, rocprofsys, timemory, trace  |
| ROCPROFSYS_TRACE_PERIODS            |                 | custom, librocprof-sys, perfetto, profile, rocprofsys, timemory, trace  |
| ROCPROFSYS_TRACE_PERIOD_CLOCK_ID    | CLOCK_REALTIME  | custom, librocprof-sys, perfetto, profile, rocprofsys, timemory, trace  |
| ROCPROFSYS_USE_PERFETTO             |      true       | backend, custom, deprecated, librocprof-sys, perfetto, rocprofsys       |
|-------------------------------------|-----------------|-------------------------------------------------------------------------|
```

Shows all runtime settings that may be tuned for perfetto

Rocprof-sys Configuration

```
$ rocprof-sys-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:

```
$ rocprof-sys-avail -h
```

Collect information for rocprof-sys-related settings using shorthand -c for --categories :

```
$ rocprof-sys-avail -c rocprofsys
```

For brief description, use the options:

```
$ rocprof-sys-avail -bd
```

ENVIRONMENT VARIABLE	
ROCPROFSYS_CAUSAL_BACKEND	Backend for call-stack sampling. See https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/performing-causal-profiling.html#backends for
ROCPROFSYS_CAUSAL_BINARY_EXCLUDE	Excludes binaries matching the list of provided regexes from causal experiments (separated by tab, semi-colon, and/or quotes (single or double))
ROCPROFSYS_CAUSAL_BINARY_SCOPE	Limits causal experiments to the binaries matching the provided list of regular expressions (separated by tab, semi-colon, and/or quotes (single or double)
ROCPROFSYS_CAUSAL_DELAY	Length of time to wait (in seconds) before starting the first causal experiment
ROCPROFSYS_CAUSAL_DURATION	Length of time to perform causal experimentation (in seconds) after the first experiment has started. After this amount of time has elapsed, no more causal
ROCPROFSYS_CAUSAL_FUNCTION_EXCLUDE	Excludes functions matching the list of provided regexes from causal experiments (separated by tab, semi-colon, and/or quotes (single or double))
ROCPROFSYS_CAUSAL_FUNCTION_SCOPE	List of <function> regex entries for causal profiling (separated by tab, semi-colon, and/or quotes (single or double))
ROCPROFSYS_CAUSAL_MODE	Perform causal experiments at the function-scope or line-scope. Ideally, use function first to locate function with highest impact and then switch to line
ROCPROFSYS_CAUSAL_RANDOM_SEED	Seed for random number generator which selects speedups and experiments -- please note that the lines selected for experimentation are not reproducible but
ROCPROFSYS_CAUSAL_SOURCE_EXCLUDE	Excludes source files or source file + lineno pair (i.e. <file> or <file>:<line>) matching the list of provided regexes from causal experiments (separated
ROCPROFSYS_CAUSAL_SOURCE_SCOPE	Limits causal experiments to the source files or source file + lineno pair (i.e. <file> or <file>:<line>) matching the provided list of regular expressions
ROCPROFSYS_CONFIG_FILE	Configuration file for rocprofiler-systems
ROCPROFSYS_ENABLED	Activation state of timemory
ROCPROFSYS_OUTPUT_PATH	Explicitly specify the output folder for results
ROCPROFSYS_OUTPUT_PREFIX	Explicitly specify a prefix for all output files
ROCPROFSYS_PAPI_EVENTS	PAPI presets and events to collect (see also: papi_avail)
ROCPROFSYS_PERFETTO_BACKEND	Specify the perfetto backend to activate. Options are: 'inprocess', 'system', or 'all'
ROCPROFSYS_PERFETTO_BUFFER_SIZE_KB	Size of perfetto buffer (in KB)
ROCPROFSYS_PERFETTO_FILL_POLICY	Behavior when perfetto buffer is full. 'discard' will ignore new entries, 'ring_buffer' will overwrite old entries
ROCPROFSYS_PROCESS_SAMPLING_DURATION	If > 0.0, time (in seconds) to sample before stopping. If less than zero, uses ROCPROFSYS_SAMPLING_DURATION
ROCPROFSYS_PROCESS_SAMPLING_FREQ	Number of measurements per second when OMNITTRACE_USE_PROCESS_SAMPLING=ON. If set to zero, uses ROCPROFSYS_SAMPLING_FREQ value
ROCPROFSYS_PROFILE	Enable timemory backend

Create a config file

Create a config file in $HOME:

```
$ rocprof-sys-avail -G $HOME/.rocprof_sys.cfg
```

To add description of all variables and settings, use:

```
$ rocprof-sys-avail -G $HOME/.rocprof_sys.cfg --all
```

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

```
<snip>
ROCPROFSYS_TRACE                        = true
ROCPROFSYS_PROFILE                      = false
ROCPROFSYS_USE_SAMPLING                 = false
ROCPROFSYS_USE_PROCESS_SAMPLING         = true
ROCPROFSYS_USE_ROCTRACER                = false
ROCPROFSYS_USE_ROCM_SMI                 = false
ROCPROFSYS_USE_KOKKOSP                  = false
ROCPROFSYS_USE_MPIP                     = true
ROCPROFSYS_USE_PID                      = true
ROCPROFSYS_USE_RCCLP                    = false
ROCPROFSYS_USE_ROCPROFILER                e
ROCPROFSYS_USE_ROCTX                    = false
<snip>
```

Contents of the config file

Declare which config file to use by setting the environment:

```
$ export ROCPROFSYS_CONFIG_FILE=/path-to/.rocprof_sys.cfg
```

AMD
together we advance_

Dynamic Instrumentation

Runtime Instrumentation

Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```
$ make –f Makefile[.cray]
```

Run the non-instrumented code on a single GPU as:

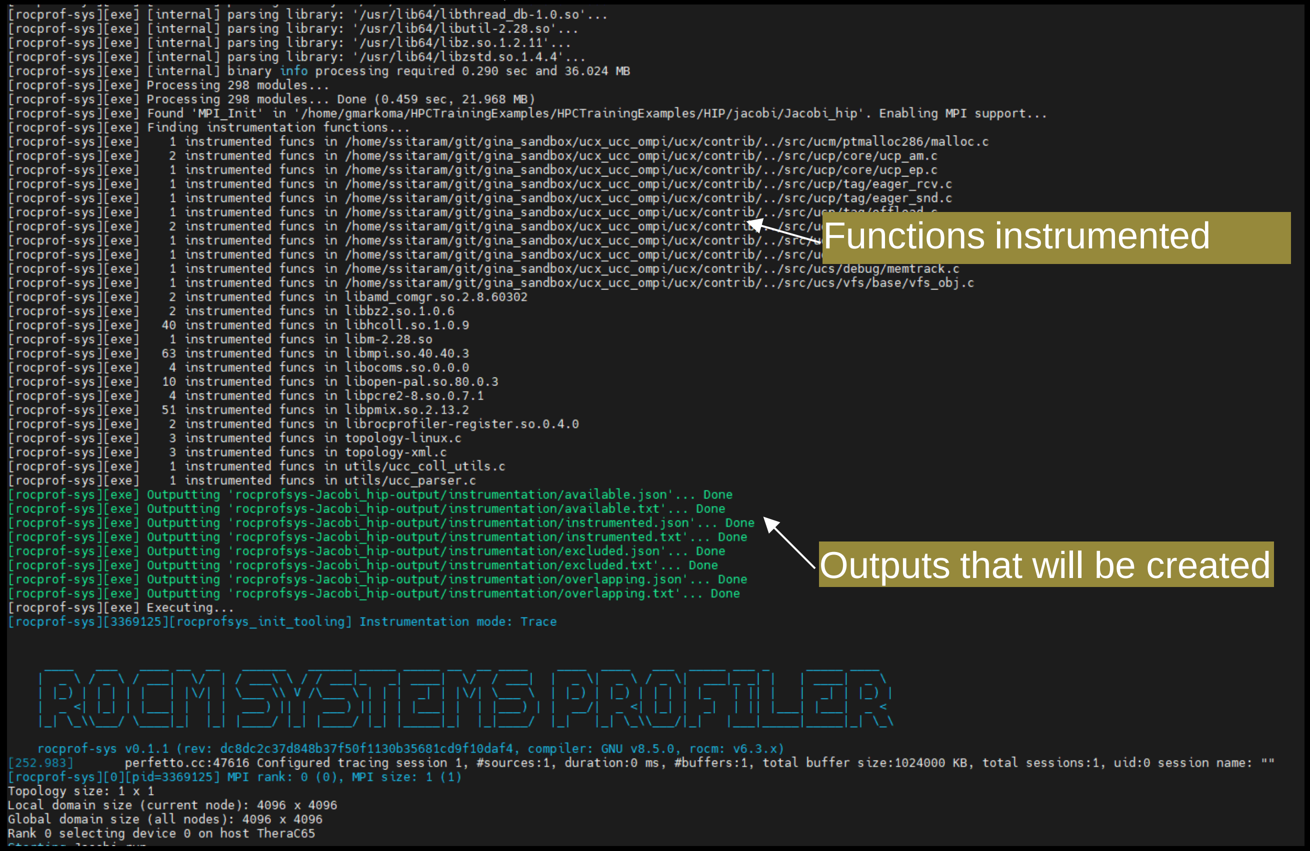
```
 $ time srun -n 1 --gpus=1 ./Jacobi_hip -g 1 1
real    0m2.115s
```

Dynamic instrumentation

```
 $ time srun -n 1 --gpus=1 rocprof-sys-instrument --
./Jacobi_hip -g 1 1

 real 2m23.742s
```

Extra time is the overhead of dyninst reading every binary that is loaded, not overhead of rocprof-sys during app execution

Parsing libraries



```
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libthread_db-1.0.so'...
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libutil-2.28.so'...
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libz.so.1.2.11'...
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libzstd.so.1.4.4'...
[rocprof-sys][exe] [internal] binary info processing required 0.290 sec and 36.024 MB
[rocprof-sys][exe] Processing 298 modules...
[rocprof-sys][exe] Processing 298 modules... Done (0.459 sec, 21.968 MB)
[rocprof-sys][exe] Found 'MPI_Init' in '/home/gmarkoma/HPCTrainingExamples/HPCTrainingExamples/HIP/jacobi/Jacobi_hip'. Enabling MPI support...
[rocprof-sys][exe] Finding instrumentation functions...
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucm/ptmalloc286/malloc.c
[rocprof-sys][exe]    2 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucp/core/ucp_am.c
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucp/core/ucp_ep.c
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucp/tag/eager_rcv.c
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucp/tag/eager_snd.c
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/u
[rocprof-sys][exe]    2 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contri
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/u
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucs/debug/memtrack.c
[rocprof-sys][exe]    1 instrumented funcs in /home/ssitaram/git/gina_sandbox/ucx_ucc_ompi/ucx/contrib/../src/ucs/vfs/base/vfs_obj.c
[rocprof-sys][exe]    2 instrumented funcs in libamd_comgr.so.2.8.60302
[rocprof-sys][exe]    2 instrumented funcs in libbz2.so.1.0.6
[rocprof-sys][exe]   40 instrumented funcs in libhcoll.so.1.0.9
[rocprof-sys][exe]    1 instrumented funcs in libm-2.28.so
[rocprof-sys][exe]   63 instrumented funcs in libmpi.so.40.40.3
[rocprof-sys][exe]    4 instrumented funcs in libocoms.so.0.0.0
[rocprof-sys][exe]   10 instrumented funcs in libopen-pal.so.80.0.3
[rocprof-sys][exe]    4 instrumented funcs in libpcre2-8.so.0.7.1
[rocprof-sys][exe]   51 instrumented funcs in libpmix.so.2.13.2
[rocprof-sys][exe]    2 instrumented funcs in librocprofiler-register.so.0.4.0
[rocprof-sys][exe]    3 instrumented funcs in topology-linux.c
[rocprof-sys][exe]    3 instrumented funcs in topology-xml.c
[rocprof-sys][exe]    1 instrumented funcs in utils/ucc_coll_utils.c
[rocprof-sys][exe]    1 instrumented funcs in utils/ucc_parser.c
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/available.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/available.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/instrumented.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/instrumented.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/excluded.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/excluded.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/overlapping.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/overlapping.txt'... Done
[rocprof-sys][exe] Executing...
[rocprof-sys][3369125][rocprofsys_init_tooling] Instrumentation mode: Trace

    rocprof-sys v0.1.1 (rev: dc8dc2c37d848b37f50f1130b35681cd9f10daf4, compiler: GNU v8.5.0, rocm: v6.3.x)
[252.983]       perfetto.cc:47616 Configured tracing session 1, #sources:1, duration:0 ms, #buffers:1, total buffer size:1024000 KB, total sessions:1, uid:0 session name: ""
[rocprof-sys][0][pid=3369125] MPI rank: 0 (0), MPI size: 1 (1)
Topology size: 1 x 1
Local domain size (current node): 4096 x 4096
Global domain size (all nodes): 4096 x 4096
Rank 0 selecting device 0 on host TheraC65
```

Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```
$ make
```

Run the non-instrumented code on a single GPU as:

```
 $ time srun -n 1 ./Jacobi_hip -g 1 1
real    0m2.115s
```

Dynamic instrumentation

```
 $ time srun -n 1 --gpus=1 rocprof-sys-instrument --
./Jacobi_hip -g 1 1

 real 2m23.742s
```

Available functions to instrument:

```
 $ srun -n 1 --gpus=1 rocprof-sys-instrument -v 1 --
simulate --print-available functions -- ./Jacobi_hip -g 1 1
```

Here, -v gives a verbose output from rocprof-sys

The simulate flag does not run the executable, but only demonstrates the available functions

```
[available] Input.hip:
[available]    [ExtractNumber][15]
[available]    [FindAndClearArgument][32]
[available]    [ParseCommandLineArguments][320]
[available]    [PrintUsage][12]

[available] JacobiIteration.hip:
[available]    [JacobiIteration][71]

[available] JacobiMain.hip:
[available]    [main][38]

[available] JacobiRun.hip:
[available]    [Jacobi_t::Run][146]

[available] JacobiSetup.hip:
[available]    [FormatNumber][44]
[available]    [Jacobi_t::ApplyTopology][215]
[available]    [Jacobi_t::CreateMesh][373]
[available]    [Jacobi_t::InitializeData][632]
[available]    [Jacobi_t::Jacobi_t][461]
[available]    [Jacobi_t::PrintResults][98]
[available]    [Jacobi_t::~Jacobi_t][171]
[available]    [PrintPerfCounter][94]
[available]    [_GLOBAL__sub_I_Jac

[available] Jacobi_hip:
[available]    [__clang_call_termi
[available]    [__device_stub__Hal
[available]    [__device_stub__JacobiIterationKernel][38]
[available]    [__device_stub__LocalLaplacianKernel][38]
[available]    [__device_stub__NormKernel1][32]
[available]    [__device_stub__NormKernel2][26]
[available]    [__do_fini][20]
[available]    [__do_init][15]
[available]    [__hip_module_ctor][20]
[available]    [__hip_module_ctor][33]
[available]    [__hip_module_dtor][8]
[available]    [__libc_csu_fini][2]
[available]    [__libc_csu_init][34]
[available]    [_dl_relocate_static_pie][2]
[available]    [_fini][4]
[available]    [_init][8]
[available]    [_start][13]
[available]    [atexit][4]
[available]    [targ227275][1]
```

Functions found in each module detected by rocprof-sys

AMD
together we advance_

Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```
$ make
```

Run the non-instrumented code on a single GPU as:

```
 $ time srun -n 1 ./Jacobi_hip -g 1 1
real    0m2.115s
```

Dynamic instrumentation

```
 $ time srun -n 1 --gpus=1 rocprof-sys-instrument --
./Jacobi_hip -g 1 1

 real 2m23.742s
```

Available functions to instrument:

```
 $ srun -n 1 --gpus=1 rocprof-sys-instrument -v 1 --
simulate --print-available functions -- ./Jacobi_hip -g 1 1
```

Custom include/exclude functions* with -I or -E, resp. For e.g:

```
 $ srun -n 1 --gpus=1 rocprof-sys-instrument -v 1 -I
'Jacobi_t::Run' 'JacobiIteration' -- ./Jacobi_hip -g 1 1
```

Include two functions to instrument

```
[rocprof-sys][exe] function: 'rocprofsys_push_trace' ... found
[rocprof-sys][exe] function: 'rocprofsys_pop_trace' ... found
[rocprof-sys][exe] function: 'rocprofsys_register_source' ... found
[rocprof-sys][exe] function: 'rocprofsys_register_coverage' ... found
[rocprof-sys][exe] function: 'rocprofsys_set_instrumented' ... found
[rocprof-sys][exe] Resolved 'librocprof-sys-dl.so' to '/opt/rocm-6.3.2/lib/librocpr
[rocprof-sys][exe] Adding main entry snippets...
[rocprof-sys][exe] Adding main exit snippets...
[rocprof-sys][exe] [function][Instrumenting] no-constraint :: 'JacobiIteration'...
[rocprof-sys][exe] [function][Instrumenting] no-constraint :: 'Jacobi_t::Run'...
[rocprof-sys][exe] [function][Instrumenting] no-constraint :: '__device_stub__JacobiIterationKernel'...
[rocprof-sys][exe] [function][Instrumenting] no-constraint :: 'fmt::v10::detail::vformat_to<char>(fmt::v10::detail::buffer<ch
[rocprof-sys][exe] [function][Instrumenting] no-constraint :: 'fmt::v10::detail::write_float<char, fmt::v10::appender, long d
[rocprof-sys][exe]    1 instrumented funcs in JacobiIteration.hip
[rocprof-sys][exe]    1 instrumented funcs in JacobiRun.hip
[rocprof-sys][exe]    1 instrumented funcs in Jacobi_hip
[rocprof-sys][exe]    2 instrumented funcs in librocprofiler-register.so.0.4.0
[rocprof-sys][exe]
^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/available.json'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/available.txt'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/instrumented.json'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/instrumented.txt'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/excluded.json'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/excluded.txt'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/overlapping.json'... Done
^[[0m^[[01;32m[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip-output/instrumentation/overlapping.txt'... Done
^[[0m[rocprof-sys][exe] Executing...
^[[0m^[[0m^[[01;34m[rocprof-sys][3371117][rocprofsys_init_tooling] Instrumentation mode: Trace
```

Only these functions are shown to be instrumented

Dynamic Instrumentation

Binary Rewrite

Binary Rewrite – Jacobi Example

Binary Rewrite

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

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

```
$ rocprof-sys-instrument -o Jacobi_hip.inst -- ./Jacobi_hip
```

This new binary will have instrumented functions

```
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libutil-2.28.so'...
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libz.so.1.2.11'...
[rocprof-sys][exe] [internal] parsing library: '/usr/lib64/libzstd.so.1.4.4'...
[rocprof-sys][exe] [internal] binary info processing required 0.341 sec and 93.808 MB
[rocprof-sys][exe] Processing 9 modules...
[rocprof-sys][exe] Processing 9 modules... Done (0.002 sec, 0.000 MB)
[rocprof-sys][exe] Found 'MPI_Init' in '/home/gmarkoma/HPCTrainingExamples/HIP/jacobi/Jacobi_hip'. Enabling MPI support...
[rocprof-sys][exe] Finding instrumentation functions...
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/available.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/available.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/instrumented.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/instrumented.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/excluded.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/excluded.txt'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/overlapping.json'... Done
[rocprof-sys][exe] Outputting 'rocprofsys-Jacobi_hip.inst-output/instrumentation/overlapping.txt'... Done
[rocprof-sys][exe]
[rocprof-sys][exe] The instrumented executable image is stored in '/home/gmarkoma/HPCTrainingExamples/HIP/jacobi/Jacobi_hip.inst'
[rocprof-sys][exe] Getting linked libraries for /home/gmarkoma/HPCTrainingExamples/HIP/jacobi/Jacobi_hip...
[rocprof-sys][exe] Consider instrumenting the relevant libraries...
[rocprof-sys][exe]
[rocprof-sys][exe]     /share/contrib-modules/openmpi/ompi5.0.3-ucc1.3.x-ucx1.16.x-rocm6.2.0/lib/libmpi.so.40
[rocprof-sys][exe]     /opt/rocm-6.2.0/lib/llvm/bin/../../../lib/libroctx64.so.4
[rocprof-sys][exe]     /opt/rocm-6.2.0/lib/llvm/bin/../../../lib/libroctracer64.so.4
[rocprof-sys][exe]     /opt/rocm-6.2.0/lib/llvm/bin/../../../lib/libamdhip64.so.6
```

Path to new instrumented binary

Subroutine Instrumentation

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

To instrument routines with 50 or more cycles, add option "-i 50" (more overhead)

AMD
together we advance_

Binary Rewrite – Jacobi Example

Binary Rewrite

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

Generating a new /library with instrumentation built-in:

```
 $ rocprof-sys-instrument -o Jacobi_hip.inst --
./Jacobi_hip
```

Run the instrumented binary:

```
 $ srun -n 1 --gpus=1 rocprof-sys-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 50 or more cycles, add option `"-i 50"` (more overhead)

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

```
    rocprof-sys v0.1.1 (rev: dc8dc2c37d848b37f50f1130b35681cd9f10daf4, compiler: GNU v8.5.0, rocm: v6.3.x)
[072.164]       perfetto.cc:47616 Configured tracing session 1, #sources:1, duration:0 ms, #buffers:1, total buffer size:1024000 KB, total sessions:1, uid:0 session name: ""
[rocprof-sys][0][pid=3371865] MPI rank: 0 (0), MPI size: 1 (1)
Topology size: 1 x 1
Local domain size (current node): 4096 x 4096
Global domain size (all nodes): 4096 x 4096
Rank 0 selecting device 0 on host TheraC65
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: 1.3976 sec.
Measured lattice updates: 12.00 GLU/s (total), 12.00 GLU/s (per process)
Measured FLOPS: 204.07 GFLOPS (total), 204.07 GFLOPS (per process)
Measured device bandwidth: 1.15 TB/s (total), 1.15 TB/s (per process)

[rocprof-sys][3371865][0][rocprofsys_finalize] finalizing...
[rocprof-sys][3371865][0][rocprofsys_finalize]
[rocprof-sys][3371865][0][rocprofsys_finalize] rocprofsys/process/3371865 : 2.849334 sec wall_clock,  984.688 MB peak_rss,  555.131 MB page_rss, 5.310000 sec cpu_clock,  186.4 % cpu_util [laps: 1]
[rocprof-sys][3371865][0][rocprofsys_finalize] rocprofsys/process/3371865/thread/0 : 2.842854 sec wall_clock, 2.244633 sec thread_cpu_clock,   79.0 % thread_cpu_util,  984.072 MB peak_rss [laps: 1]
[rocprof-sys][3371865][0][rocprofsys_finalize] rocprofsys/process/3371865/thread/2 : 0.000018 sec wall_clock, 0.000017 sec thread_cpu_clock,   99.8 % thread_cpu_util,    0.000 MB peak_rss [laps: 1]
[rocprof-sys][3371865][0][rocprofsys_finalize] rocprofsys/process/3371865/thread/3 : 2.634946 sec wall_clock, 0.000759 sec thread_cpu_clock,    0.0 % thread_cpu_util,  965.000 MB peak_rss [laps: 1]
[rocprof-sys][3371865][0][rocprofsys_finalize] rocprofsys/process/3371865/thread/4 : 2.394117 sec wall_clock, 0.012569 sec thread_cpu_clock,    0.5 % thread_cpu_util,  950.828 MB peak_rss [laps: 1]
[rocprof-sys][3371865][0][rocprofsys_finalize]
[rocprof-sys][3371865][0][rocprofsys_finalize] Finalizing perfetto...
[rocprofiler-systems][3371865][perfetto]> Outputting '/home/gmarkoma/HPCTrainingExamples/HIP/jacobi/rocprofsys-Jacobi_hip.inst-output/2025-04-29_06.08/perfetto-trace-0.proto' (5902.60 KB / 5.90 MB / 0.01 GB)... Done
[rocprofiler-systems][3371865][metadata]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-29_06.08/metadata-0.json' and 'rocprofsys-Jacobi_hip.inst-output/2025-04-29_06.08/functions-0.json'
[rocprof-sys][3371865][0][rocprofsys_finalize] Finalized: 0.560843 sec wall_clock,    0.000 MB peak_rss,    1.122 MB page_rss, 0.560000 sec cpu_clock,   99.8 % cpu_util
[075.581]       perfetto.cc:49205 Tracing session 1 ended, total sessions:0
```

Generates traces for application run

List of Instrumented GPU Functions

Declare ROCPROFSYS_PROFILE = true in your cfg file

```
$ cat rocprofsys-Jacobi_hip.inst-output/2023-03-15_13.57/roctracer-0.txt
```

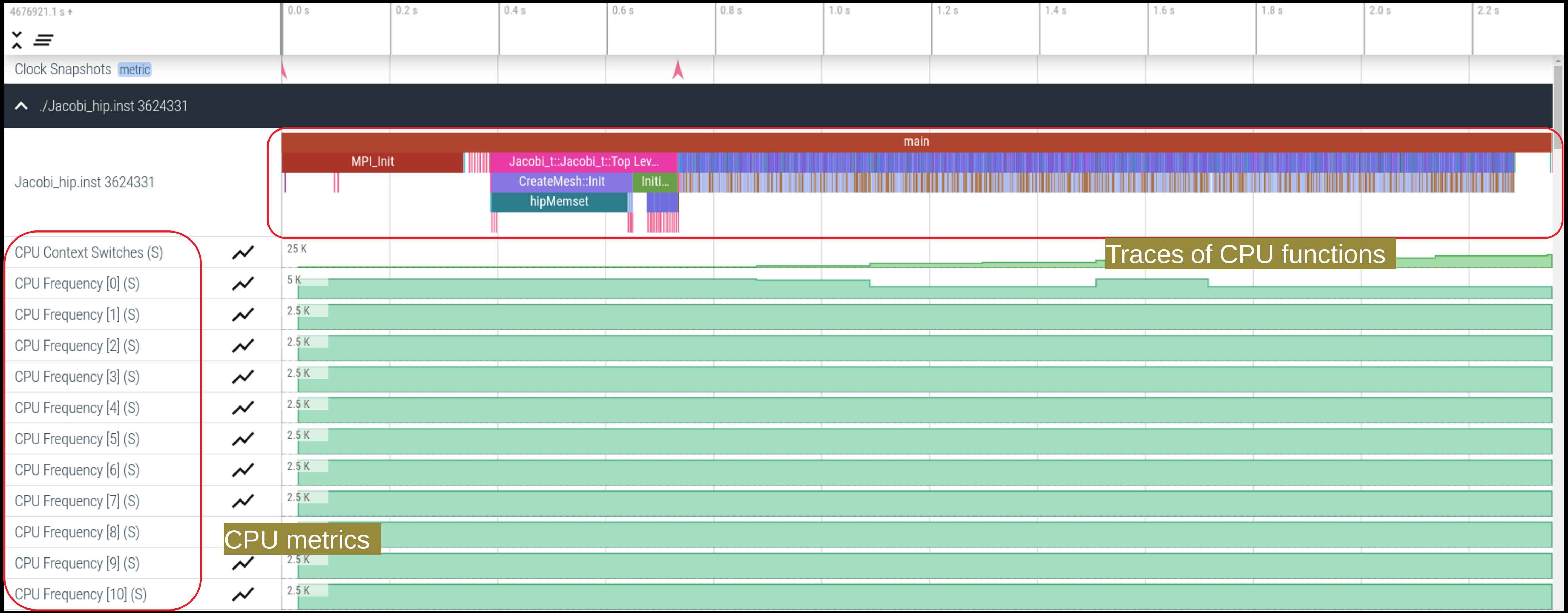
```
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                     ROCM TRACER (ACTIVITY API)                                                                         |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                          LABEL                                                | COUNT  | DEPTH  | METRIC    | UNITS  | SUM      | MEAN     | % SELF |
|-----------------------------------------------------------------------------------------------|--------|--------|-----------|--------|----------|----------|--------|
| |0>>> pthread_create                                                                          |      1 |      0 | roctracer | sec    | 0.000353 | 0.000353 |    0.0 |
| |1>>> |_start_thread                                                                          |      1 |      1 | roctracer | sec    | 2.344864 | 2.344864 |  100.0 |
| |0>>> hipInit                                                                                 |      1 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipGetDeviceCount                                                                       |      1 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipSetDevice                                                                            |      1 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipHostMalloc                                                                           |      3 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipMalloc                                                                               |      7 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipMemset                                                                               |      1 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipStreamCreate                                                                         |      2 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipMemcpy                                                                               |   1005 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> |_LocalLaplacianKernel(int, int, int, double, double, double const*, double*)           |    999 |      1 | roctracer | sec    | 0.279368 | 0.000280 |  100.0 |
| |0>>> |_HaloLaplacianKernel(int, int, int, double, double, double const*, double const*, double*) |    990 |      1 | roctracer | sec    | 0.014761 | 0.000015 |  100.0 |
| |0>>> |_JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*) |    959 |      1 | roctracer | sec    | 0.531156 | 0.000554 |  100.0 |
| |0>>> |_NormKernel1(int, double, double, double const*, double*)                              |    997 |      1 | roctracer | sec    | 0.430196 | 0.000431 |  100.0 |
| |0>>> |_NormKernel2(int, double const*, double*)                                              |    999 |      1 | roctracer | sec    | 0.004342 | 0.000004 |  100.0 |
| |0>>> hipEventCreate                                                                          |      2 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipLaunchKernel                                                                         |   5002 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> |_JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*) |      1 |      1 | roctracer | sec    | 0.000552 | 0.000552 |  100.0 |
| |0>>> |_NormKernel1(int, double, double, double const*, double*)                              |      1 |      1 | roctracer | sec    | 0.000425 | 0.000425 |  100.0 |
| |0>>> hipDeviceSynchronize                                                                    |   1001 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> |_NormKernel1(int, double, double, double const*, double*)                              |      2 |      1 | roctracer | sec    | 0.000850 | 0.000425 |  100.0 |
| |0>>> |_NormKernel2(int, double const*, double*)                                              |      1 |      1 | roctracer | sec    | 0.000004 | 0.000004 |  100.0 |
| |0>>> |_HaloLaplacianKernel(int, int, int, double, double, double const*, double const*, double*) |      9 |      1 | roctracer | sec    | 0.000133 | 0.000015 |  100.0 |
| |0>>> |_JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*) |     40 |      1 | roctracer | sec    | 0.022204 | 0.000555 |  100.0 |
| |0>>> |_LocalLaplacianKernel(int, int, int, double, double, double const*, double*)           |      1 |      1 | roctracer | sec    | 0.000281 | 0.000281 |  100.0 |
| |0>>> hipEventRecord                                                                          |   2000 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipStreamSynchronize                                                                    |   2000 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipEventElapsedTime                                                                     |   1000 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> |_HaloLaplacianKernel(int, int, int, double, double, double const*, double const*, double*) |      1 |      1 | roctracer | sec    | 0.000015 | 0.000015 |  100.0 |
| |0>>> hipFree                                                                                 |      4 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
| |0>>> hipHostFree                                                                             |      2 |      0 | roctracer | sec    | 0.000000 | 0.000000 |    0.0 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Roctracer-0.txt shows duration of HIP API calls and GPU kernels

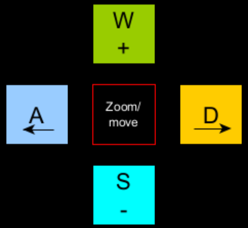
Visualizing Trace

Use Perfetto

Copy `perfetto-trace-0.proto` to your laptop, go to https://ui.perfetto.dev/, Click "Open trace file", select perfetto-trace-0.proto

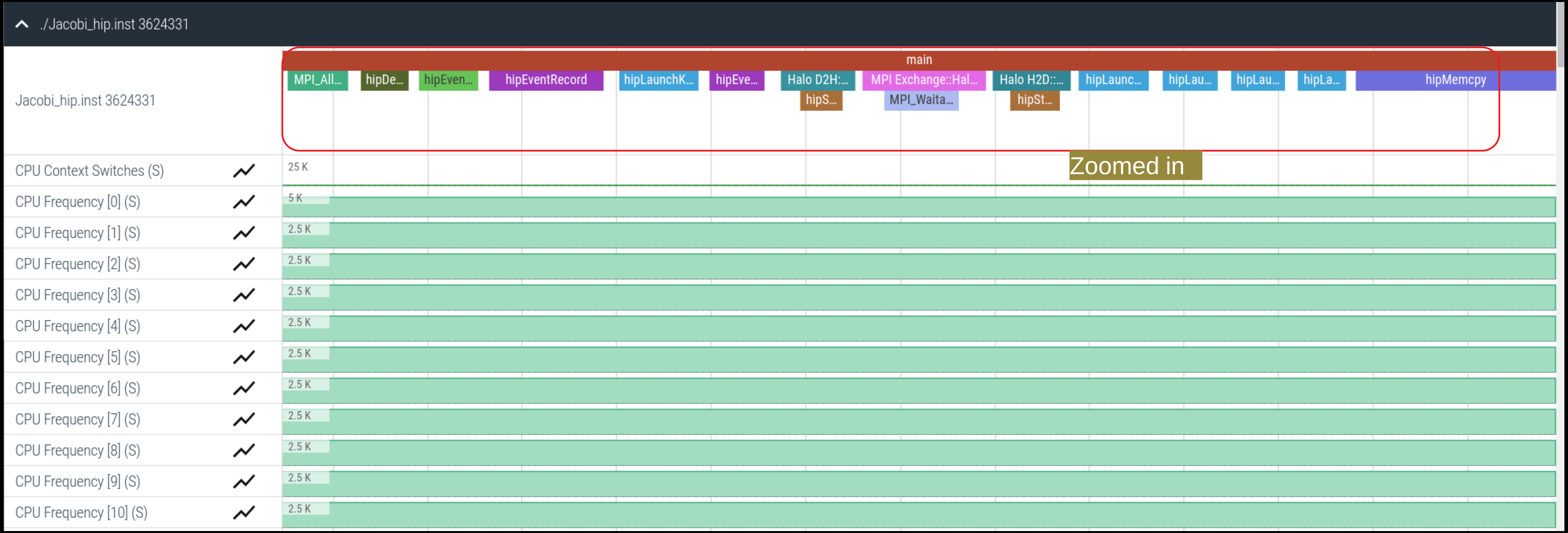


Visualizing Trace



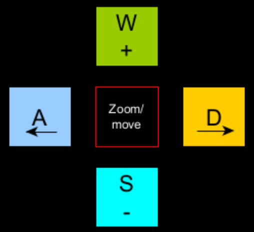
Use Perfetto

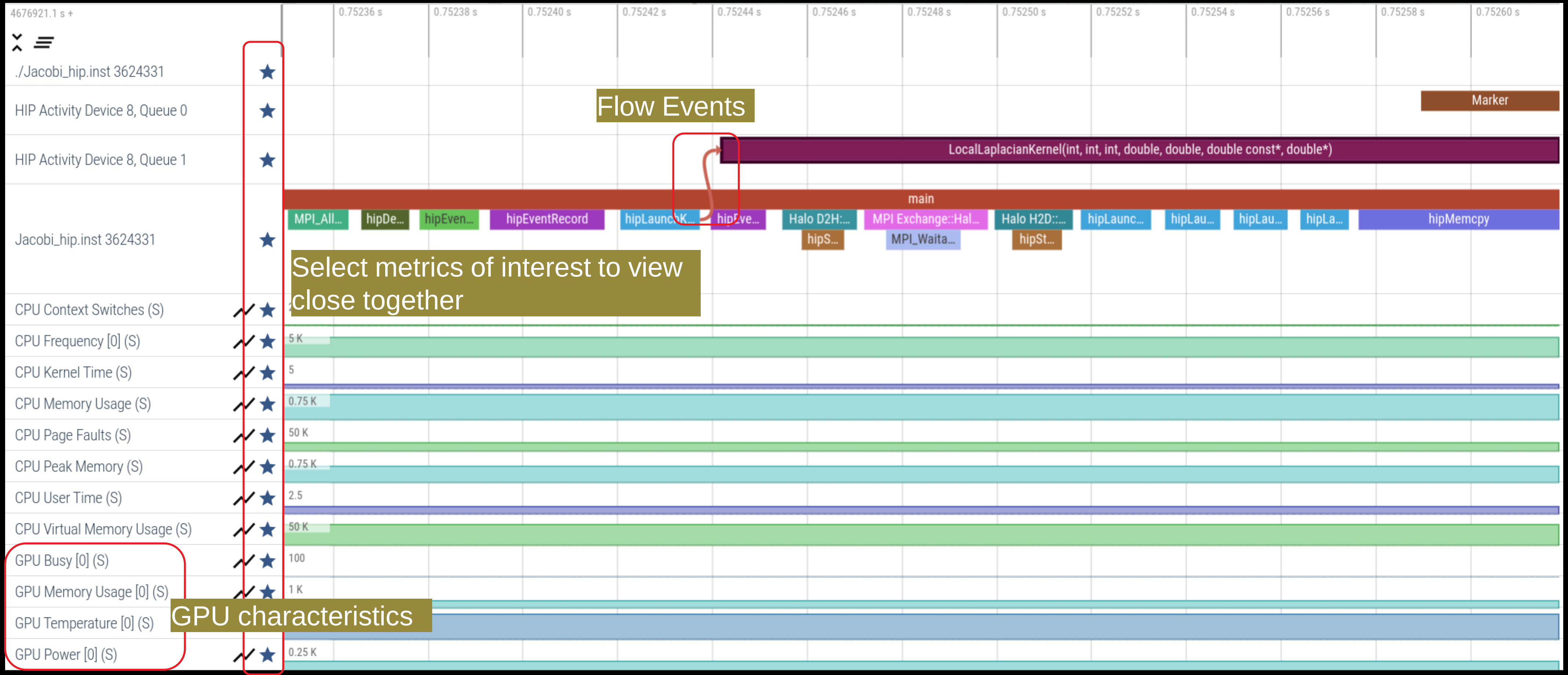
Zoom in to investigate regions of interest



Visualizing Trace

Use Perfetto
Zoom in to investigate regions of interest





Larger Traces with Perfetto

- There is a memory limit in the Chrome browser. There is a way to read in the trace for the browser before starting it up.

Linux

- curl -LO https://get.perfetto.dev/trace_processor
- chmod +x ./trace_processor
- ./trace_processor --httpd <path to trace file>
- Open up Chrome browser and go to https://ui.perfetto.dev
- When prompted, click on "Yes, use loaded trace"

Windows

- Open up https://get.perfetto.dev/trace_processor in a browser to download the python script
- py trace_processor --httpd <trace file>
  - You may need to download and install python on your windows system
- Open up Chrome browser and go to https://ui.perfetto.dev
- When prompted, click on "Yes, use loaded trace"

Hardware Counters

[Public]

Hardware Counters – List All

```
$ srun -n 1 rocprof-sys-avail --all
```

Components, Categories

COMPONENT	AVAILABLE	VALUE_TYPE	STRING_IDS	FILENAME	DESCRIPTION	CATEGORY
allinea_map	false	void	"allinea", "allinea_map", "forge"		Controls the AllineaMAP sampler.	category::external, os::supports_linux, t...
caliper_marker	false	void	"cali", "caliper", "caliper_marker"		Generic forwarding of markers to Caliper ...	category::external, os::supports_unix, tp...
caliper_config	false	void	"caliper_config"		Caliper configuration manager.	category::external, os::supports_unix, tp...
caliper_loop_marker	false	void	"caliper_loop_marker"		Variant of caliper_marker with support fo...	category::external, os::supports_unix, tp...
cpu_clock	true	long	"cpu_clock"	cpu_clock	Total CPU time spent in both user- and ke...	project::timemory, category::timing, os::...
cpu_util	true	std::pair<long, long>	"cpu_util", "cpu_utilization"	cpu_util	Percentage of CPU-clock time divided by w...	project::timemory, category::timing, os::...
craypat_counters	false	std::vector<unsigned long, std::allocato...	"craypat_counters"	craypat_counters	Names and value of any counter events tha...	category::external, os::supports_linux, t...

Environment Variables

ENVIRONMENT VARIABLE	VALUE	DATA TYPE	DESCRIPTION	CATEGORIES
ROCPROFSYS_CAUSAL_BACKEND	auto	string	Backend for call-stack sampling. See https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/performing-causal-profiling.html#backends for more info. If set to "auto", rocprof-sys will attempt to use the perf backend and fallback on the timer backend if unavailable	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_BINARY_EXCLUDE		string	Excludes binaries matching the list of provided regexes from causal experiments (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_BINARY_SCOPE	%MAIN%	string	Limits causal experiments to the binaries matching the provided list of regular expressions (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_DELAY	0	double	Length of time to wait (in seconds) before starting the first causal experiment	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_DURATION	0	double	Length of time to perform causal experimentation (in seconds) after the first experiment has started. After this amount of time has elapsed, no more causal experiments will be performed and the application will continue without any overhead from causal profiling. Any value <= 0 means until the application completes	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_FUNCTION_EXCLUDE		string	Excludes functions matching the list of provided regexes from causal experiments (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_FUNCTION_SCOPE		string	List of <function> regex entries for causal profiling (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_MODE	function	string	Perform causal experiments at the function-scope or line-scope. Ideally, use function first to locate function with highest impact and then switch to line mode + ROCPROFSYS_CAUSAL_FUNCTION_SCOPE set to the function being targeted.	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_RANDOM_SEED	0	unsigned long	Seed for random number generator which selects speedups and experiments -- please note that the lines selected for experimentation are not reproducible but the speedup selection is. If set to zero, std::random_device{}() will be used.	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_SOURCE_EXCLUDE		string	Excludes source files or source file + lineno pair (i.e. <file> or <file>:<line>) matching the list of provided regexes from causal experiments (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CAUSAL_SOURCE_SCOPE		string	Limits causal experiments to the source files or source file + lineno pair (i.e. <file> or <file>:<line>) matching the provided list of regular expressions (separated by tab, semi-colon, and/or quotes (single or double))	analysis, causal, custom, librocprof-sys, rocprofsys
ROCPROFSYS_CONFIG_FILE	/home/gmarkoma/test.cfg	string	Configuration file for rocprofiler-systems	config, core, librocprof-sys, rocprofsys, timemory
ROCPROFSYS_ENABLED	true	bool	Activation state of timemory	core, timemory
ROCPROFSYS_OUTPUT_PATH	rocprofsys-%tag%-output	string	Explicitly specify the output folder for results	filename, io, librocprof-sys, rocprofsys, timemory

CPU Hardware Counters

HARDWARE COUNTER	AVAILABLE	DESCRIPTION
CPU		
PAPI_L1_DCM	true	Level 1 data cache misses
PAPI_L1_ICM	false	Level 1 instruction cache misses
PAPI_L2_DCM	true	Level 2 data cache misses
PAPI_L2_ICM	true	Level 2 instruction cache misses
PAPI_L3_DCM	false	Level 3 data cache misses
PAPI_L3_ICM	false	Level 3 instruction cache misses
PAPI_L1_TCM	[illegible]	Level 1 cache misses

perf::CYCLES	true	PERF_COUNT_HW_CPU_CYCLES
perf::CYCLES:u=0	true	perf::CYCLES + monitor at user level
perf::CYCLES:k=0	true	perf::CYCLES + monitor at kernel level
perf::CYCLES:h=0	true	perf::CYCLES + monitor at hypervisor level
perf::CYCLES:period=0	true	perf::CYCLES + sampling period
perf::CYCLES:freq=0	true	perf::CYCLES + sampling frequency (Hz)
perf::CYCLES:precise=0	true	perf::CYCLES + precise event sampling
perf::CYCLES:excl=0	true	perf::CYCLES + exclusive access

GPU Hardware Counters

TCC_NORMAL_WRITEBACK_sum:device=0	true	Number of writebacks due to requests that...
TCC_ALL_TC_OP_WB_WRITEBACK_sum:device=0	true	Number of writebacks due to all TC_OP wri...
TCC_NORMAL_EVICT_sum:device=0	true	Number of evictions due to requests that ...
TCC_ALL_TC_OP_INV_EVICT_sum:device=0	true	Number of evictions due to all TC_OP inva...
TCC_EA_RDREQ_DRAM_sum:device=0	true	Number of TCC/EA read requests (either 32...
TCC_EA_WRREQ_DRAM_sum:device=0	true	Number of TCC/EA write requests (either 3...
FETCH_SIZE:device=0	true	The total kilobytes fetched from the vide...
WRITE_SIZE:device=0	true	The total kilobytes written to the video ...
WRITE_REQ_32B:device=0	true	The total number of 32-byte effective mem...
GPUBusy:device=0	true	The percentage of time GPU was busy.
Wavefronts:device=0	[illegible]	Total wavefronts.
VALUInsts:device=0	[illegible]	The average number of vector ALU instruct...
SALUInsts:device=0	true	The average number of scalar ALU instruct...
SFetchInsts:device=0	true	The average number of scalar fetch instru...
GDSInsts:device=0	true	The average number of GDS read or GDS wri...
MemUnitBusy:device=0	true	The percentage of GPUTime the memory unit...
ALUStalledByLDS:device=0	true	The percentage of GPUTime ALU units are s...

A very small subset of the counters shown here

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
L2CacheHit	The percentage of fetch, write, atomic, and other instructions that hit the data in L2 cache
MemUnitBusy	The percentage of GPUTime the memory unit is active. The result includes the stall time
MemUnitStalled	The percentage of GPUTime the memory unit is stalled
WriteUnitStalled	The percentage of GPUTime the write unit is stalled

Modify config file

Create a config file in $HOME:

```
$ rocprof-sys-avail -G $HOME/.rocprofsys.cfg
```

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

```
…
ROCPROFSYS_ROCM_EVENTS = GPUBusy:device=0,
Wavefronts:device=0, MemUnitBusy:device=0
…
```

To profile desired metrics for all participating GPUs:

```
…
ROCPROFSYS_ROCM_EVENTS = GPUBusy, Wavefronts,
MemUnitBusy
…
```

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

Execution with Hardware Counters

(after modifying cfg file to set up ROCPROFSYS_ROCM_EVENTS with GPU metrics)

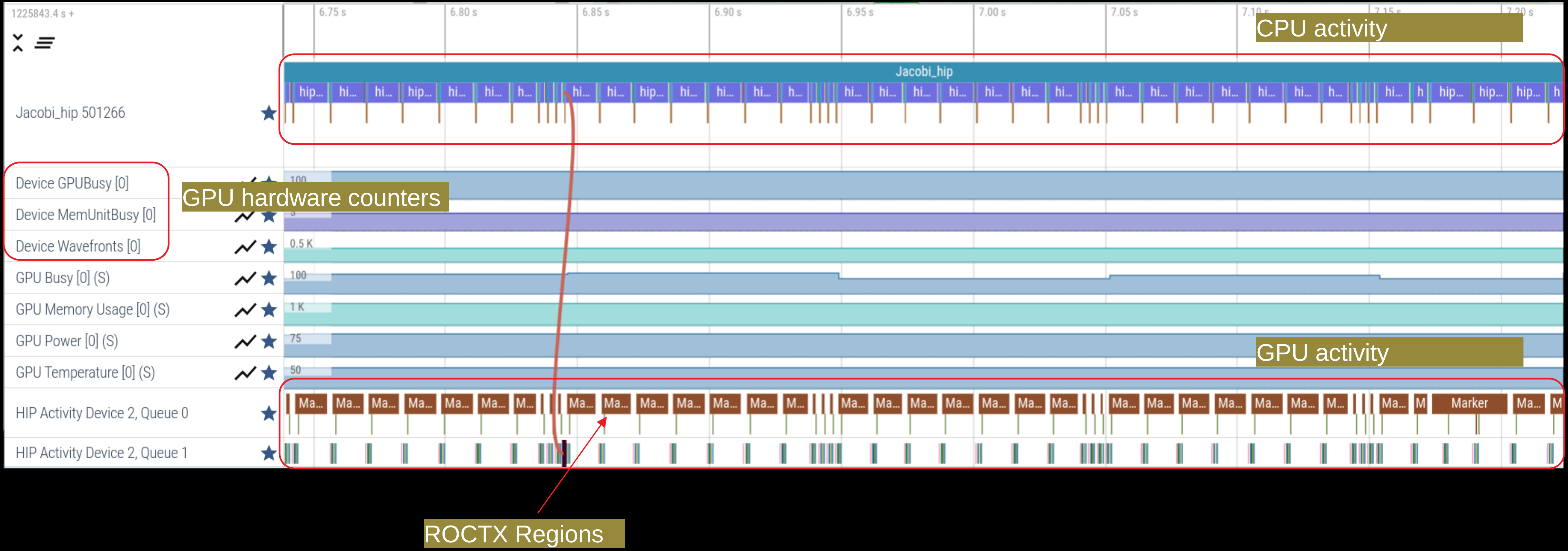
```
$ srun -n 1 rocprof-sys-run -- ./Jacobi_hip.inst -g 1 1
```

```
[rocprof-sys][2704005][0][rocprofsys_finalize] Finalizing perfetto...
[rocprofiler-systems][2704005][perfetto]> Outputting '/home/gmarkoma/HPCTrainingExamples/HIP/jacobi/rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/perfetto-trace-0.proto' (7379.91 KB / 7.38 MB / 0.01 GB)... Done
[rocprofiler-systems][2704005][rocprof-device-0-GPUBusy]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-GPUBusy-0.json'
[rocprofiler-systems][2704005][rocprof-device-0-GPUBusy]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-GPUBusy-0.txt'
[rocprofiler-systems][2704005][rocprof-device-0-Wavefronts]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-Wavefronts-0.json'
[rocprofiler-systems][2704005][rocprof-device-0-Wavefronts]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-Wavefronts-0.txt'
[rocprofiler-systems][2704005][rocprof-device-0-MemUnitBusy]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-MemUnitBusy-0.json'
[rocprofiler-systems][2704005][rocprof-device-0-MemUnitBusy]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/rocprof-device-0-MemUnitBusy-0.txt'
[rocprofiler-systems][2704005][wall_clock]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/wall_clock-0.json'
[rocprofiler-systems][2704005][wall_clock]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/wall_clock-0.txt'
[rocprofiler-systems][2704005][roctracer]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/roctracer-0.json'
[rocprofiler-systems][2704005][roctracer]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/roctracer-0.txt'
[rocprofiler-systems][2704005][metadata]> Outputting 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/metadata-0.json' and 'rocprofsys-Jacobi_hip.inst-output/2025-04-30_09.35/functions-0.json'
[rocprof-sys][2704005][0][rocprofsys_finalize] Finalized: 0.458214 sec wall_clock,    0.000 MB peak_rss,   10.424 MB page_rss, 0.450000 sec cpu_clock,   98.2 % cpu_util
[841.882]       perfetto.cc:49205 Tracing session 1 ended, total sessions:0
```

GPU hardware counters

AMD
together we advance_

Visualization with Hardware Counters



AMD
together we advance_

Tracing Multiple Ranks

Profiling Multiple MPI Ranks – Jacobi Example

Binary Rewrite

Generating a new /library with instrumentation built-in:

```
$ rocprof-sys-instrument -o Jacobi_hip.inst --
./Jacobi_hip
```

Run the instrumented binary with 2 ranks:

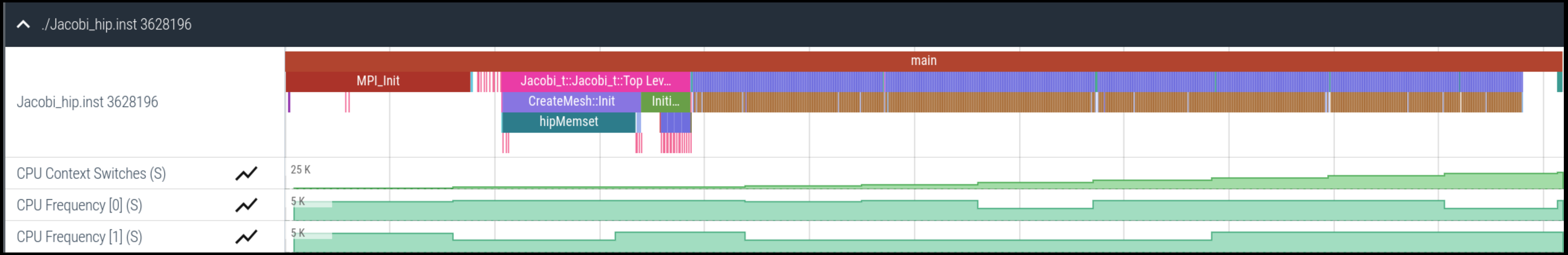
```
$ srun -n 2 rocprof-sys-run --./Jacobi_hip.inst -g 2
1
```

```
[omnitrace][3628199][perfetto]> Outputting '/home/ssitaram/git/HPCTrainingExamples/HIP/jacobi/omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/perfetto-trace-1.proto'
[perfetto]> Outputting '/home/ssitaram/git/HPCTrainingExamples/HIP/jacobi/omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/perfetto-trace-0.proto' (7856.71 KB / 7.86 M

[omnitrace][3628199][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/wall_clock-1.json'
[omnitrace][3628196][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/wall_clock-0.json'
[omnitrace][3628199][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/wall_clock-1.txt'
[omnitrace][3628196][wall_clock]> Outputting 'omnitrace-Jacobi_hip.inst-output/2023-03-15_18.02/wall_clock-0.txt'
```

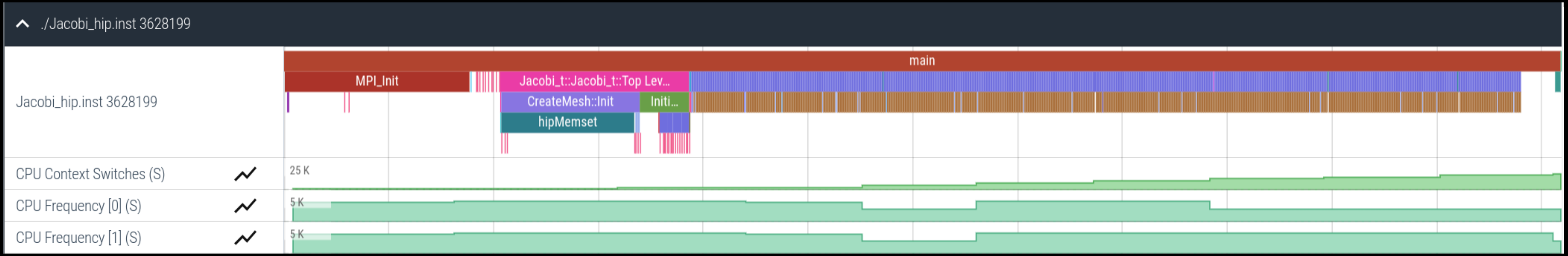
All output files are generated for each rank

Visualizing Traces from Multiple Ranks - Separately

MPI 0



MPI 1



AMD
together we advance_

Visualizing Traces from Multiple Ranks - Combined

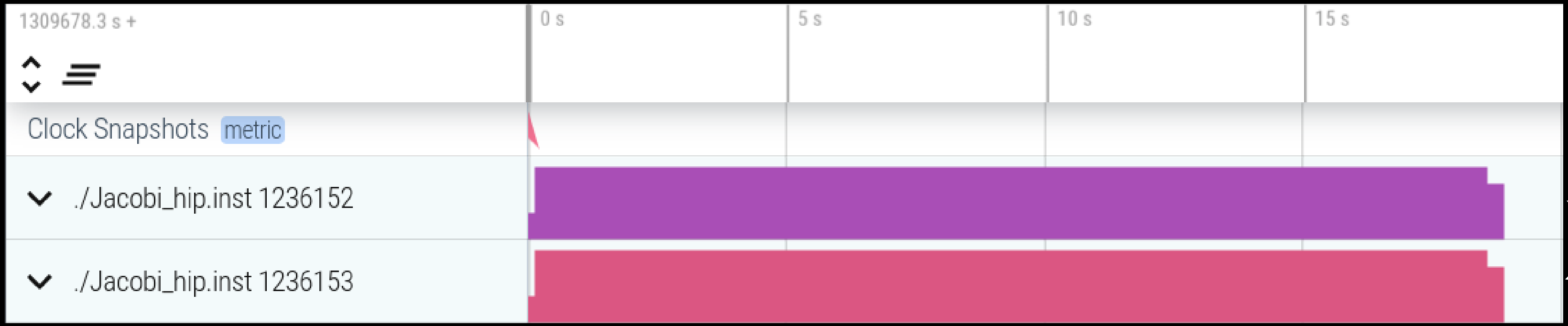
Merge Perfetto

Use the following command to merge and concatenate multiple traces:

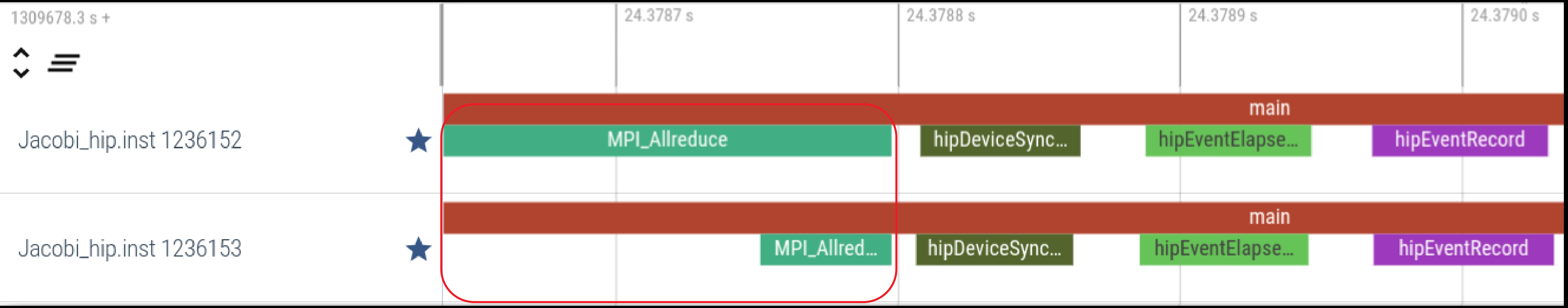
```
$ cat perfetto-trace-0.proto perfetto-trace-1.proto > allprocesses.proto
```

It seems there is an issue with newer Perfetto to visualize all the MPI processes

Try to visualize through: https://ui.perfetto.dev/v46.0-35b3d9845/#!/



Two processes seen in combined trace file



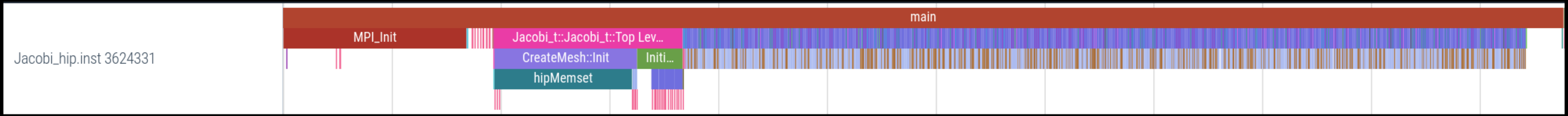
Zooming in helps understand load balance issues

AMD together we advance_

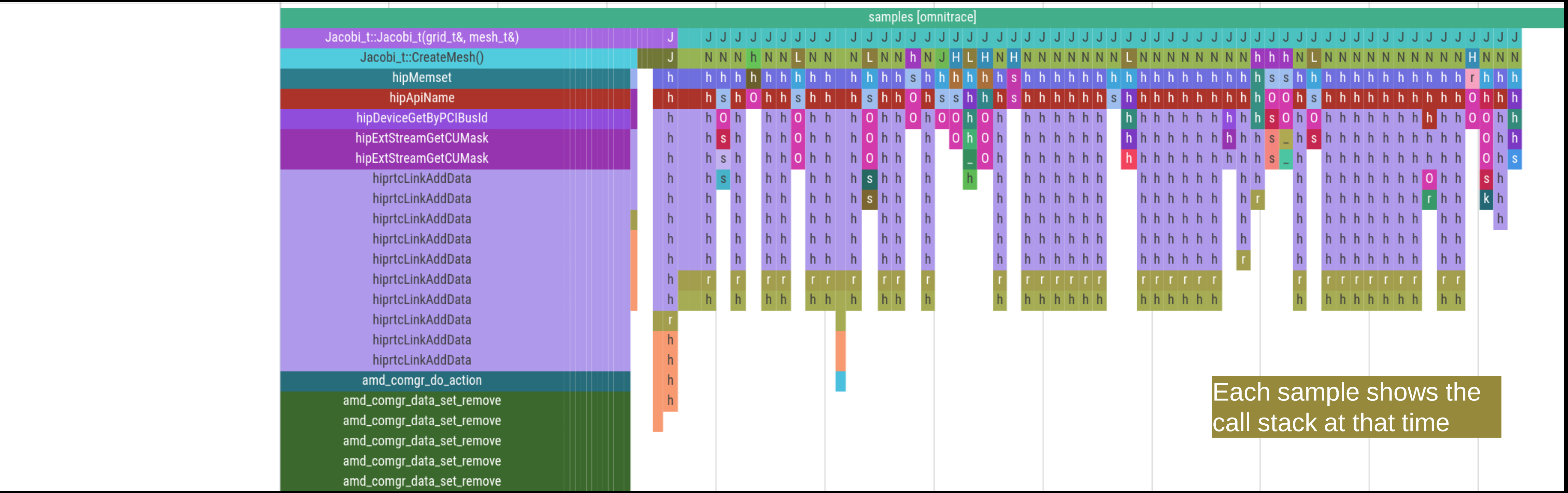
Statistical Sampling

Sampling Call-Stack (I)

`ROCPROFSYS_USE_SAMPLING = false`



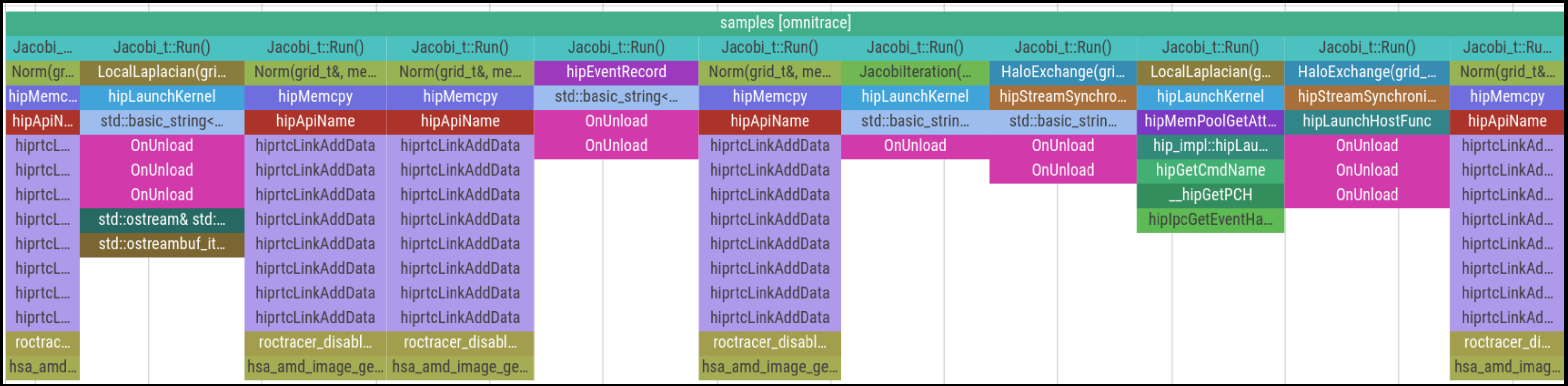
`ROCPROFSYS_USE_SAMPLING = true; ROCPROFSYS_SAMPLING_FREQ = 100` (100 samples per second)



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

Sampling Call-Stack (II)

Zoom in call-stack sampling



Sampling data is annotated with (S)

Other Features

Kernel Durations

```
$ cat rocprofsys-Jacobi_hip.inst-output/2023-03-15_13.57/wall_clock-0.txt
```

If you do not see a wall_clock.txt dumped by rocprof-sys, try modify the config file `$HOME/.rocprofsys.cfg` and enable `ROCPROFSYS_ENABLED`:

```
…
ROCPROFSYS_USE_PERFETTO                            = true
ROCPROFSYS_PROFILE                                 = true
ROCPROFSYS_USE_SAMPLING                            = false
…
```

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 |
```

Text file is for quick reference. JSON output is easy to script for and can be read by Hatchet, a Python package (https://hatchet.readthedocs.io/en/latest/)

Kernel Durations (flat profile)

Edit in your rocprofsys.cfg:

```
ROCPROFSYS_PROFILE          = true
ROCPROFSYS_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	-	-	-	-	-	-	-	-	-

AMD
together we advance_

User API

Rocprof-sys provides an API to control the instrumentation

API Call	Description
`int` **`rocprofsys_user_start_trace(void)`**	Enable tracing on this thread and all subsequently created threads
`int` **`rocprofsys_user_stop_trace(void)`**	Disable tracing on this thread and all subsequently created threads
`int` **`rocprofsys_user_start_thread_trace(void)`**	Enable tracing on this specific thread. Does not apply to subsequently created threads
`int` **`rocprofsys_user_stop_thread_trace(void)`**	Disable tracing on this specific thread. Does not apply to subsequently created threads
`int` **`rocprofsys_user_push_region(void)`**	Start user defined region
`int` **`rocprofsys_user_pop_region(void)`**	End user defined region, FILO (first in last out) is expected

All the API calls: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/using-rocprof-sys-api.html

OpenMP®

We use the example rocprofiler-systems/examples/openmp/
Build the code with CMake:

```
$ cmake -B build
```

Use the openmp-lu binary, which can be executed with:

```
$ export OMP_NUM_THREADS=4
$ srun -n 1 -c 4 ./openmp-lu
```

Create a new instrumented binary:

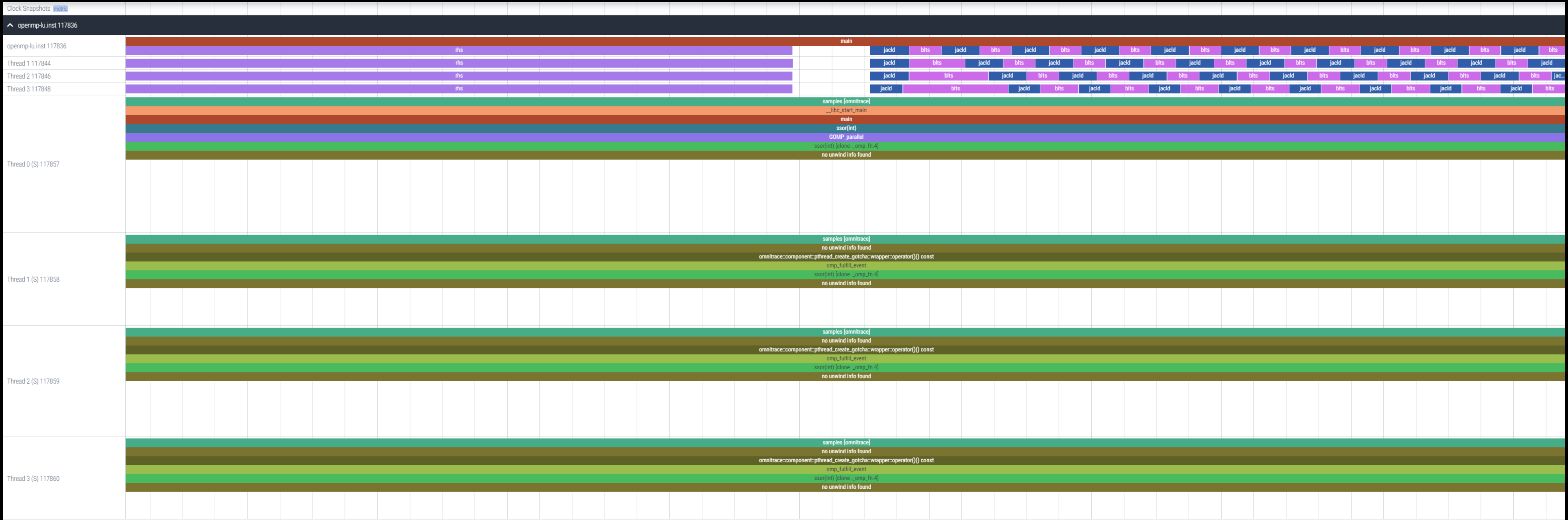
```
$ srun -n 1 rocprof-sys-instrument -o openmp-lu.inst -
- ./openmp-lu
```

Execute the new binary:

```
$ srun -n 1 -c 4 rocprof-sys-run -- ./openmp-lu.inst
```

```
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                     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    | 1.096702 | 1.096702 | 1.096702 | 1.096702 | 0.000000 | 0.000000 |    9.2 |
| |0>>> |_pthread_create   |      3 |      1 | wall_clock | sec    | 0.002931 | 0.000977 | 0.000733 | 0.001420 | 0.000000 | 0.000385 |    0.0 |
| |3>>>   |_start_thread   |      1 |      2 | wall_clock | sec    | 2.451520 | 2.451520 | 2.451520 | 2.451520 | 0.000000 | 0.000000 |   57.7 |
| |3>>>     |_erhs         |      1 |      3 | wall_clock | sec    | 0.001906 | 0.001906 | 0.001906 | 0.001906 | 0.000000 | 0.000000 |  100.0 |
| |3>>>     |_rhs          |    153 |      3 | wall_clock | sec    | 0.229893 | 0.001503 | 0.001410 | 0.001893 | 0.000000 | 0.000116 |  100.0 |
| |3>>>     |_jacld        |   3473 |      3 | wall_clock | sec    | 0.170568 | 0.000049 | 0.000047 | 0.000135 | 0.000000 | 0.000005 |  100.0 |
| |3>>>     |_blts         |   3473 |      3 | wall_clock | sec    | 0.232512 | 0.000067 | 0.000040 | 0.000959 | 0.000000 | 0.000034 |  100.0 |
| |3>>>     |_jacu         |   3473 |      3 | wall_clock | sec    | 0.166229 | 0.000048 | 0.000046 | 0.000148 | 0.000000 | 0.000005 |  100.0 |
| |3>>>     |_buts         |   3473 |      3 | wall_clock | sec    | 0.236484 | 0.000068 | 0.000041 | 0.000391 | 0.000000 | 0.000031 |  100.0 |
| |2>>>   |_start_thread   |      1 |      2 | wall_clock | sec    | 2.452309 | 2.452309 | 2.452309 | 2.452309 | 0.000000 | 0.000000 |   58.1 |
| |2>>>     |_erhs         |      1 |      3 | wall_clock | sec    | 0.001895 | 0.001895 | 0.001895 | 0.001895 | 0.000000 | 0.000000 |  100.0 |
| |2>>>     |_rhs          |    153 |      3 | wall_clock | sec    | 0.229776 | 0.001502 | 0.001410 | 0.001893 | 0.000000 | 0.000115 |  100.0 |
| |2>>>     |_jacld        |   3473 |      3 | wall_clock | sec    | 0.204609 | 0.000059 | 0.000057 | 0.000152 | 0.000000 | 0.000006 |  100.0 |
| |2>>>     |_blts         |   3473 |      3 | wall_clock | sec    | 0.192986 | 0.000056 | 0.000047 | 0.000358 | 0.000000 | 0.000026 |  100.0 |
| |2>>>     |_jacu         |   3473 |      3 | wall_clock | sec    | 0.199029 | 0.000057 | 0.000055 | 0.000188 | 0.000000 | 0.000007 |  100.0 |
| |2>>>     |_buts         |   3473 |      3 | wall_clock | sec    | 0.198972 | 0.000057 | 0.000048 | 0.000372 | 0.000000 | 0.000026 |  100.0 |
| |1>>>   |_start_thread   |      1 |      2 | wall_clock | sec    | 2.453072 | 2.453072 | 2.453072 | 2.453072 | 0.000000 | 0.000000 |   58.6 |
| |1>>>     |_erhs         |      1 |      3 | wall_clock | sec    | 0.001905 | 0.001905 | 0.001905 | 0.001905 | 0.000000 | 0.000000 |  100.0 |
| |1>>>     |_rhs          |    153 |      3 | wall_clock | sec    | 0.229742 | 0.001502 | 0.001410 | 0.001894 | 0.000000 | 0.000115 |  100.0 |
| |1>>>     |_jacld        |   3473 |      3 | wall_clock | sec    | 0.206418 | 0.000059 | 0.000057 | 0.000934 | 0.000000 | 0.000016 |  100.0 |
| |1>>>     |_blts         |   3473 |      3 | wall_clock | sec    | 0.186097 | 0.000054 | 0.000047 | 0.000344 | 0.000000 | 0.000023 |  100.0 |
| |1>>>     |_jacu         |   3473 |      3 | wall_clock | sec    | 0.198689 | 0.000057 | 0.000055 | 0.000186 | 0.000000 | 0.000006 |  100.0 |
| |1>>>     |_buts         |   3473 |      3 | wall_clock | sec    | 0.192470 | 0.000055 | 0.000048 | 0.000356 | 0.000000 | 0.000022 |  100.0 |
| |0>>> |_erhs             |      1 |      1 | wall_clock | sec    | 0.001961 | 0.001961 | 0.001961 | 0.001961 | 0.000000 | 0.000000 |  100.0 |
| |0>>> |_rhs              |    153 |      1 | wall_clock | sec    | 0.229889 | 0.001503 | 0.001410 | 0.001891 | 0.000000 | 0.000116 |  100.0 |
| |0>>> |_jacld            |   3473 |      1 | wall_clock | sec    | 0.208903 | 0.000060 | 0.000057 | 0.000359 | 0.000000 | 0.000017 |  100.0 |
| |0>>> |_blts             |   3473 |      1 | wall_clock | sec    | 0.172646 | 0.000050 | 0.000047 | 0.000822 | 0.000000 | 0.000020 |  100.0 |
| |0>>> |_jacu             |   3473 |      1 | wall_clock | sec    | 0.202130 | 0.000058 | 0.000055 | 0.000350 | 0.000000 | 0.000016 |  100.0 |
| |0>>> |_buts             |   3473 |      1 | wall_clock | sec    | 0.176975 | 0.000051 | 0.000048 | 0.000377 | 0.000000 | 0.000016 |  100.0 |
| |0>>> |_pintgr           |      1 |      1 | wall_clock | sec    | 0.000054 | 0.000054 | 0.000054 | 0.000054 | 0.000000 | 0.000000 |  100.0 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
```

OpenMP® Visualization



AMD
together we advance_

Python™

The rocprofsys Python package is installed in /path/rocprofsys_install/lib/pythonX.Y/site-packages/rocprofsys

Setup the environment:

```
$ export PYTHONPATH=/path/rocprofsys/lib/python/site-packages/:${PYTHONPATH}
```

We use the Fibonacci example in rocprofiler-systems/examples/python/source.py

Execute the python program with:

```
$ rocprof-sys-python ./external.py
```

Profiled data is dumped in output directory:

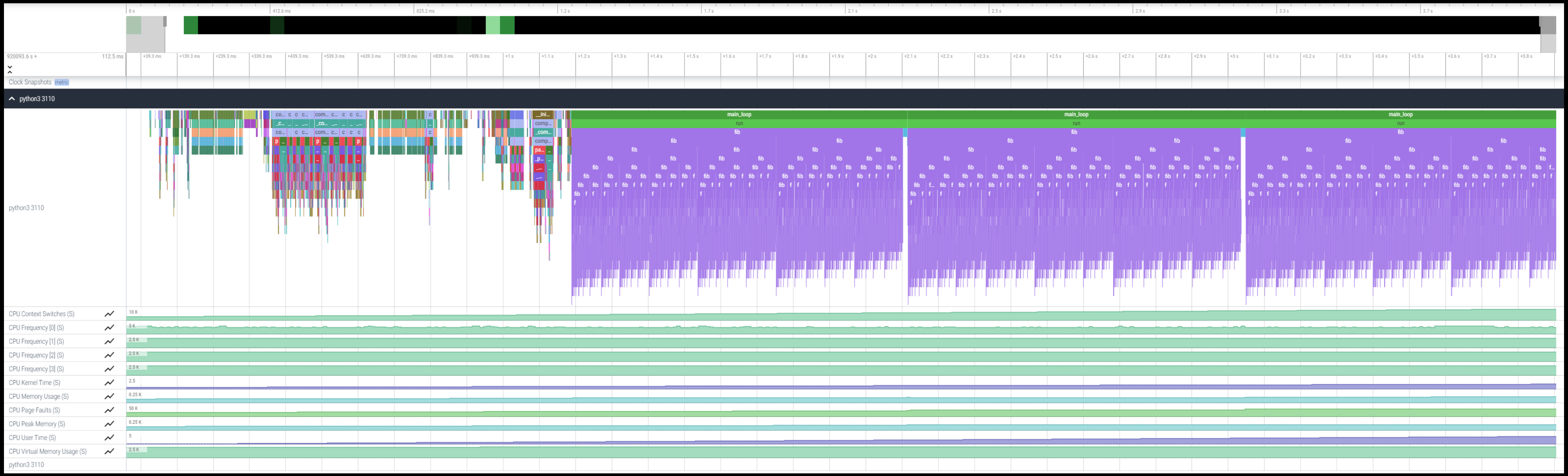
```
$ cat rocprofsys-source-output/timestamp/wall_clock.txt
```

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

LABEL	COUNT	DEPTH	METRIC	UNITS	SUM	MEAN	MIN	MAX	VAR	STDDEV	% SELF
|0>>> main_loop	3	0	wall_clock	sec	2.786075	0.928692	0.926350	0.932130	0.000009	0.003042	0.0
|0>>> |_run	3	1	wall_clock	sec	2.785799	0.928600	0.926250	0.932037	0.000009	0.003043	0.0
|0>>> |_fib	3	2	wall_clock	sec	2.750104	0.916701	0.914454	0.919577	0.000007	0.002619	0.0
|0>>> |_fib	6	3	wall_clock	sec	2.749901	0.458317	0.348962	0.567074	0.013958	0.118145	0.0
|0>>> |_fib	12	4	wall_clock	sec	2.749511	0.229126	0.133382	0.350765	0.006504	0.080650	0.0
|0>>> |_fib	24	5	wall_clock	sec	2.748734	0.114531	0.050867	0.217030	0.002399	0.048977	0.1
|0>>> |_fib	48	6	wall_clock	sec	2.747118	0.057232	0.019302	0.134596	0.000806	0.028396	0.1
|0>>> |_fib	96	7	wall_clock	sec	2.743922	0.028583	0.007181	0.083350	0.000257	0.016026	0.2
|0>>> |_fib	192	8	wall_clock	sec	2.737564	0.014258	0.002690	0.051524	0.000079	0.008887	0.5
|0>>> |_fib	384	9	wall_clock	sec	2.724966	0.007096	0.000973	0.031798	0.000024	0.004865	0.9
|0>>> |_fib	768	10	wall_clock	sec	2.699251	0.003515	0.000336	0.019670	0.000007	0.002637	1.9
|0>>> |_fib	1536	11	wall_clock	sec	2.648006	0.001724	0.000096	0.012081	0.000002	0.001417	3.9
|0>>> |_fib	3072	12	wall_clock	sec	2.545260	0.000829	0.000016	0.007461	0.000001	0.000758	8.0
|0>>> |_fib	6078	13	wall_clock	sec	2.342276	0.000385	0.000016	0.004669	0.000000	0.000404	16.0
|0>>> |_fib	10896	14	wall_clock	sec	1.967475	0.000181	0.000015	0.002752	0.000000	0.000218	28.6
|0>>> |_fib	15060	15	wall_clock	sec	1.404069	0.000093	0.000015	0.001704	0.000000	0.000123	43.6
|0>>> |_fib	14280	16	wall_clock	sec	0.791873	0.000055	0.000015	0.001044	0.000000	0.000076	58.3
|0>>> |_fib	8826	17	wall_clock	sec	0.330189	0.000037	0.000015	0.000620	0.000000	0.000050	70.9
|0>>> |_fib	3456	18	wall_clock	sec	0.096120	0.000028	0.000015	0.000380	0.000000	0.000034	81.0
|0>>> |_fib	822	19	wall_clock	sec	0.018294	0.000022	0.000015	0.000209	0.000000	0.000024	88.9
|0>>> |_fib	108	20	wall_clock	sec	0.002037	0.000019	0.000016	0.000107	0.000000	0.000015	94.9
|0>>> |_fib	6	21	wall_clock	sec	0.000104	0.000017	0.000016	0.000019	0.000000	0.000001	100.0
|0>>> |_inefficient	3	2	wall_clock	sec	0.035450	0.011817	0.010096	0.012972	0.000002	0.001519	95.8
|0>>> |__sum	3	3	wall_clock	sec	0.001494	0.000498	0.000440	0.000537	0.000000	0.000051	100.0

Python documentation: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/profiling-python-scripts.html

Visualizing Python™ Perfetto Tracing



AMD
together we advance_

Kokkos

Rocprofsys can instrument Kokkos applications too.

Edit the $HOME/.rocprofsys.cfg file and enable Kokkos:

```
...
ROCPROFSYS_USE_KOKKOSP = true
...
```

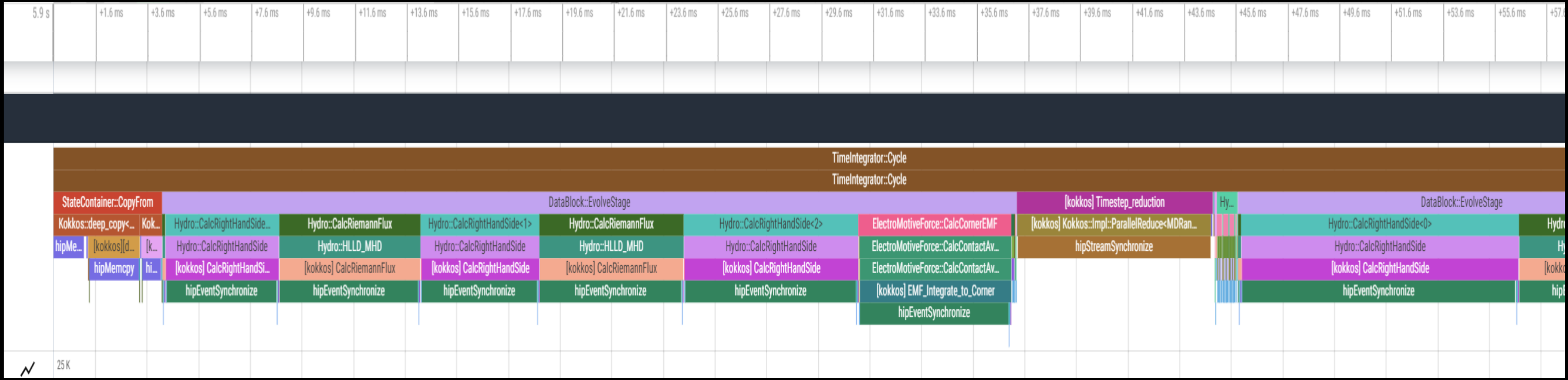
Profiling with rocprofsys produces *kokkos*.txt files:

```
$ cat kokkos_memory0.txt
```

```
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>   |_[kokkos][deep_copy] Host=DataBlock_A2_mirror HIP=DataBlock_A2                              |     1 |     2 | kokkos_memory | MB     |   142 |   142 |   100 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, pre view equality check  |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, pre view equality check        |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>   |_[kokkos][deep_copy] Host=DataBlock_dV_mirror HIP=DataBlock_dV                              |     1 |     2 | kokkos_memory | MB     |   140 |   140 |   100 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, pre view equality check  |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, pre view equality check        |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>> |_DataBlockHost::SyncToDevice()                                                                |     1 |     1 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>   |_[kokkos][deep_copy] HIP=Hydro_Vc Host=Hydro_Vc_mirror                                      |     1 |     2 | kokkos_memory | MB     |  1124 |  1124 |   100 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, pre view equality check  |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, pre view equality check        |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>   |_[kokkos][deep_copy] HIP=Hydro_InvDt Host=Hydro_InvDt_mirror                                |     1 |     2 | kokkos_memory | MB     |   140 |   140 |   100 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, pre view equality check  |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, pre view equality check        |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>   |_[kokkos][deep_copy] HIP=Hydro_Vs Host=Hydro_Vs_mirror                                      |     1 |     2 | kokkos_memory | MB     |   426 |   426 |   100 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, pre view equality check  |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, pre view equality check        |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos][dev0] Kokkos::deep_copy: copy between contiguous views, post deep copy fence      |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
| |0>>>     |_[kokkos] Kokkos::deep_copy: copy between contiguous views, post deep copy fence            |     1 |     3 | kokkos_memory | MB     |     0 |     0 |     0 |
```

Visualizing Kokkos with Perfetto Trace

- Visualize `perfetto-trace-0.proto` (with sampling enabled)



AMD
together we advance_

Other Executables

- `rocprof-sys-sample`
  - For sampling with low overhead, use `rocprof-sys-sample`
  - Use `rocprof-sys-sample --help` to get relevant options
  - Settings in the rocprofsys config file will be used by `rocprof-sys-sample`
  - Example invocation to get a flat tracing profile on Host and Device (`-PTHD`), excluding all components (`-E all`) and including only `rocm-smi, roctracer, rocprofiler` and `roctx` components (`-I ...`)

```
mpirun -np 1 rocprof-sys-sample -PTHD -E all -I rocm-smi -I roctracer -I rocprofiler -I roctx -- ./Jacobi_hip -g 1 1
```

- `rocprof-sys-causal`
  - Invokes causal profiling
- `rocprof-sys-critical-trace`
  - Post-processing tool for critical-trace data output by rocprof-sys

Current documentation: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/reference/development-guide.html#executables

Tips & Tricks

- My Perfetto timeline seems weird how can I check the clock skew?
  - Set ROCPROFSYS_VERBOSE=1 or higher for verbose mode and it will print the timestamp skew
- It takes too long to map rocm-smi samples to kernels.
  - Temporarily set ROCPROFSYS_USE_ROCM_SMI=OFF
- What is the best way to profile multi-process runs?
  - Use rocprofsys's binary rewrite (-o) option to instrument the binary first, run the instrumented binary with mpirun/srun
- If you are doing binary rewrite and you do not get information about kernels, set:
  - HSA_TOOLS_LIB=librocprof-sys.so in the env. and set ROCPROFSYS_USE_ROCTRACER=ON in the cfg file
- My HIP application hangs in different points, what do I do?
  - Try to set HSA_ENABLE_INTERRUPT=0 in the environment, this changes how HIP runtime is notified when GPU kernels complete
- My Perfetto trace is too big, can I decrease it?
  - Yes, declare ROCPROFSYS_PERFETTO_ANNOTATIONS to false
- I want to remove the many rows of CPU frequency lines from the Perfetto trace
  - Declare the ROCPROFSYS_USE_PROCESS_SAMPLING = false

Summary

- Rocprof-sys is a powerful tool to understand CPU + GPU activity
  - Ideal for an initial look at how an application runs
- Leverages several other tools and combines their data into a comprehensive output file
  - Some tools used are AMD uProf, rocprof, rocm-smi, roctracer, perf, etc.
- Easy to visualize traces in Perfetto
- Includes several features:
  - Dynamic Instrumentation either at Runtime or using Binary Rewrite
  - Statistical Sampling for call-stack info
  - Process sampling, monitoring of system metrics during application run
  - Causal Profiling
  - Critical Path Tracing

Introduction to Rocprof-compute

and Hierarchical Roofline on AMD Instinct™ MI200/MI300 GPUs

AMD
together we advance_

ssh <you user>@aac6.amd.com -p 7001

https://hackmd.io/@gmarkoma/cug2025-AMDGPUProfiling#Rocprof-compute

https://github.com/amd/HPCTrainingExamples/OmniperfExamples

Background – AMD Profilers

ROC-profiler (rocprofv3)

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

	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

Rocprof-sys

Trace collection
- Comprehensive trace collection
  - CPU
  - GPU

Supports
- CPU copy
- HIP API
- HSA API
- GPU Kernels
- OpenMP®
- MPI
- Kokkos
- p-threads
- multi-GPU

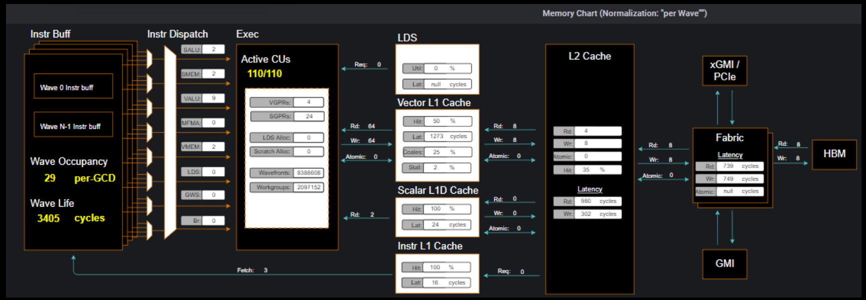
Visualisation
- Traces visualized with Perfetto

Rocprof-compute

Performance Analysis
- Automated collection of hardware counters
  - Analysis
  - Visualisation

Supports
- Speed of Light
- Memory chart
- Rooflines
- Kernel comparison

Visualisation
- With Grafana or standalone GUI



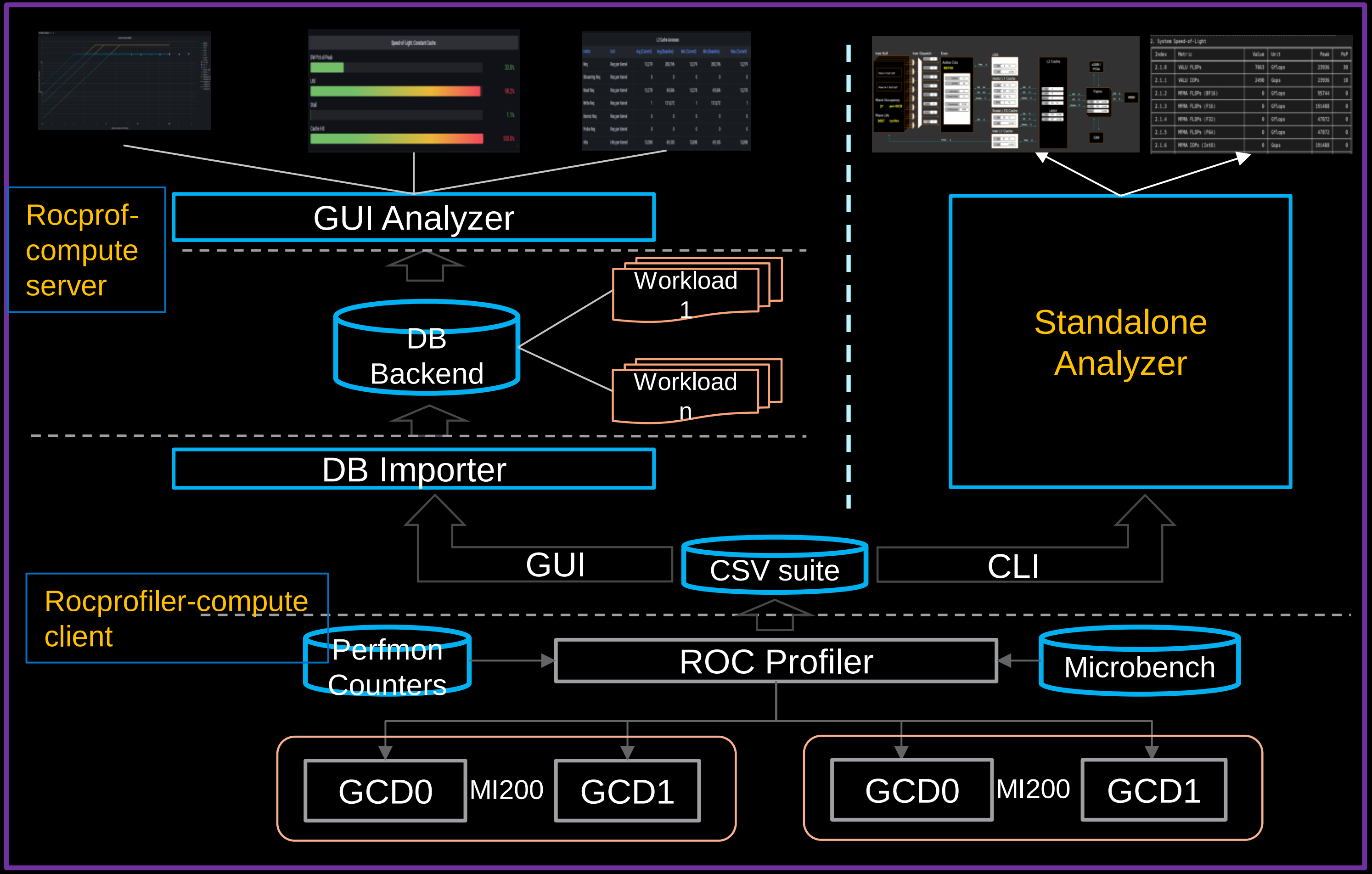
Rocprof-compute: Automated Collection of Hardware Counters and Analysis

AMD Product
- Repository: https://github.com/ROCm/rocprofiler-compute
- Part of ROCm stack after 6.3.0
- Built on top of ROC-profiler

Integrated Performance Analyzer for AMD GPUs
- Speed-of-Light | Roofline | Memory chart | Baseline comparison
- Sub-system performance analysis
- LDS | vL1D | L2 Cache | HBM
- Shader Compute | Wavefront | Instruction mix | Latencies

INSTINCT™ Support
- MI200 | MI100

User Interfaces
- Grafana™ GUI | Standalone GUI | Command Line (CLI)



Refer to current documentation for recent updates

Rocprof-compute

- Rocprof-compute is an integrated performance analyzer for AMD GPUs built on ROCprofiler
- Rocprof-compute executes the code many times to collect various hardware counters (over 100 counters default behavior)
- Using specific filtering options (kernel, dispatch ID, metric group), the overhead of profiling can be reduced
- Roofline analysis is supported on MI200 and MI300 GPUs
- Rocprof-compute shows many panels of metrics based on hardware counters, we will show a few here
- Typical Rocprof-compute workflows:
  - Profile + Analyze with CLI or visualize with standalone GUI
  - Profile + Import to database and visualize with Grafana
- Rocprof-compute targets MI100, MI200, and MI300 and future generation AMD GPUs
- Rocprof-compute requires to use just 1 MPI process

Rocprof-compute modes

- **Profile**
  - Target application is launched using AMD ROC-profiler
  - Kernels | Dispatches | IP Blocks
- **Analyze**
  - Profiled data is loaded to rocprof-compute CLI
  - Immediate access to metrics | Lightweight standalone GUI
- **Databas e**
  - Profiled data is imported to Grafana™ database
  - Grafana™ GUI is based on MongoDB | Interact with saved workload database

Basic command-line syntax:

Profile:

```
$ rocprof-compute profile -n workload_name [profile options]
                     [roofline options] -- <CMD> <ARGS>
```

Analyze:

```
$ rocprof-compute analyze -p
<path/to/workloads/workload_name/mi200/>
```

To use a lightweight standalone GUI with CLI analyzer:

```
$ rocprof-compute analyze -p
<path/to/workloads/workload_name/mi200/> --gui
```

Database:

```
$ rocprof-compute database <interaction type> [connection
options]
```

For more information or help use -h/--help/? flags:

```
$ rocprof-compute profile --help
```

Documentation: https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/

Rocprof-compute profiling

We use the example sample/vcopy.cpp from the Rocprof-compute installation

```
$ wget https://raw.githubusercontent.com/ROCm/rocprofiler-compute/refs/heads/develop/sample/vcopy.cpp
```

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Rocprof-compute:

```
$ rocprof-compute profile -n vcopy_all -- ./vcopy -n 1048576 -b 256
…
-------------
Profile only
-------------

Path:  /pfs/lustrep4/scratch/project_462000075/markoman/…
Target:  mi200
Command:  ./vcopy 1048576 256
Kernel Selection:  None
Dispatch Selection:  None
IP Blocks: All
```

A new directory will be created called `workloads/vcopy_all`

Note: Rocprof-compute executes the code as many times as required to collect all HW metrics. Use kernel/dispatch filters especially when trying to collect roofline analysis.

Rocprof-compute analyze

We use the example sample/vcopy.cpp from the Rocprof-compute installation

```
$ wget https://raw.githubusercontent.com/ROCm/rocprofiler-compute/refs/heads/develop/sample/vcopy.cpp
```

Compile with hipcc:

```
$ hipcc --offload-arch=gfx90a -o vcopy vcopy.cpp
```

Profile with Rocprof-compute:

```
$ rocprof-compute profile -n vcopy_all -- ./vcopy -n 1048576 -b 256
```

A new directory will be created called `workloads/vcopy_all`

Analyze the profiled workload:

```
$ rocprof-compute analyze -p workloads/vcopy_all/mi200/ &> vcopy_analyze.txt
```

0. Top Stat

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pc
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	341123.00	341123.00	341123.00	100.0

2. System Speed-of-Light

Index	Metric	Value	Unit	Peak	PoP
2.1.0	VALU FLOPs	0.00	Gflop	23936.0	0.0
2.1.1	VALU IOPs	89.14	Giop	23936.0	0.37242200388114116
2.1.2	MFMA FLOPs (BF16)	0.00	Gflop	95744.0	0.0
2.1.3	MFMA FLOPs (F16)	0.00	Gflop	191488.0	0.0
2.1.4	MFMA FLOPs (F32)	0.00	Gflop	47872.0	0.0
2.1.5	MFMA FLOPs (F64)	0.00	Gflop	47872.0	0.0
2.1.6	MFMA IOPs (Int8)	0.00	Giop	191488.0	0.0
2.1.7	Active CUs	58.00	Cus	110	52.72727272727273
2.1.8	SALU Util	3.69	Pct	100	3.6862586934167525
2.1.9	VALU Util	5.90	Pct	100	5.895531580380328
2.1.10	MFMA Util	0.00	Pct	100	0.0
2.1.11	VALU Active Threads/Wave	32.71	Threads	64	51.10526315789473
2.1.12	IPC - Issue	0.98	Instr/cycle	5	19.576640831930312

7.1 Wavefront Launch Stats

Index	Metric	Avg	Min	Max	Unit
7.1.0	Grid Size	1048576.00	1048576.00	1048576.00	Work items
7.1.1	Workgroup Size	256.00	256.00	256.00	Work items
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts
7.1.3	Saved Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.4	Restored Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.5	VGPRs	44.00	44.00	44.00	Registers
7.1.6	SGPRs	48.00	48.00	48.00	Registers
7.1.7	LDS Allocation	0.00	0.00	0.00	Bytes
7.1.8	Scratch Allocation	16496.00	16496.00	16496.00	Bytes

Rocprof-compute Analyze

- Execute rocprof-compute analyze –h to see various options
- Use specific IP block (-b)

Top kernels:

```
$ srun -n 1 --gpus 1 rocprof-compute analyze -p workloads/vcopy_all/mi200/ -b 0
```

IP Block of wavefronts

```
$ srun -n 1 --gpus 1 rocprof-compute analyze -p workloads/vcopy_all/mi200/ -b 7.1.2
```

```
--------------------------------------------------------------------------------
0. Top Stat
```

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	20960.00	20960.00	20960.00	100.00

```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Index	Metric	Avg	Min	Max	Unit
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts

Rocprof-compute analyze

To see available options and usage instructions:

```
 $ rocprof-compute analyze -h
...

Help:
  -h, --help                      show this help message and exit

General Options:
  -v, --version                   show program's version number and exit
  -V, --verbose                   Increase output verbosity (use multiple times for higher levels)
  -q, --quiet                     Reduce output and run quietly.
  -s, --specs                     Print system specs and exit.

Analyze Options:
  -p  [ ...], --path  [ ...]                    Specify the raw data root dirs or desired results directory.
  --list-stats                                  List all detected kernels and kernel dispatches.
  --list-metrics                                List all available metrics for analysis on specified arch:
                                                   gfx90a
                                                   gfx942
  -k  [ ...], --kernel  [ ...]                  Specify kernel id(s) from --list-stats for filtering.
  -d  [ ...], --dispatch  [ ...]                Specify dispatch id(s) for filtering.
  -b  [ ...], --block  [ ...]                   Specify hardware block/metric id(s) from --list-metrics for filtering.
  --gpu-id  [ ...]                              Specify GPU id(s) for filtering.
  -o , --output                                 Specify an output file to save analysis results.
  --gui [GUI]                                   Activate a GUI to interate with rocprofiler-compute metrics.
                                                Optionally, specify port to launch application (DEFAULT: 8050)

Advanced Options:
  --random-port                                 Randomly generate a port to launch GUI application.
                                                Registered Ports range inclusive (1024-49151).
  --max-stat-num                                Specify the maximum number of stats shown in "Top Stats" tables (DEFAULT: 10)
--decimal                                     Specify desired decimal precision of analysis results. (DEFAULT: 2)
  --config-dir                                  Specify the directory of customized configs.
  --save-dfs                                    Specify the dirctory to save analysis dataframe csv files.
  --cols  [ ...]                                Specify column indices to display.
  -g                                            Debug single metric.
  --dependency                                  List the installation dependency.
  --kernel-verbose                              Specify Kernel Name verbose level 1-5. Lower the level, shorter the kernel name. (DEFAULT: 5) (DISABLE: 5)
  --specs-correction                            Specify the specs to correct.
  --list-nodes                                  Multi-node option: list all node names.
  --nodes [ ...]                                Multi-node option: filter with node names. Enable it without node names means ALL.
```

Easy things you can check

- Are all the CUs being used?
  - If not, more parallelism is required (for most of the cases)
- Are all the VGPRs being spilled?
  - Try smaller workgroup sizes
- Is the code Integer limited?
  - Try reducing the integer ops, usually in the index calculation

[Public]

Rocprof-compute analyze with standalone GUI

We use the example sample/vcopy.cpp from the Rocprof-compute installation folder:

```
$ wget https://raw.githubusercontent.com/ROCm/rocprofiler-compute/refs/heads/develop/sample/vcopy.cpp
```

Compile with hipcc:

```
$ hipcc --offload-arch=gfx90a -o vcopy vcopy.cpp
```

Profile with Rocprof-compute:

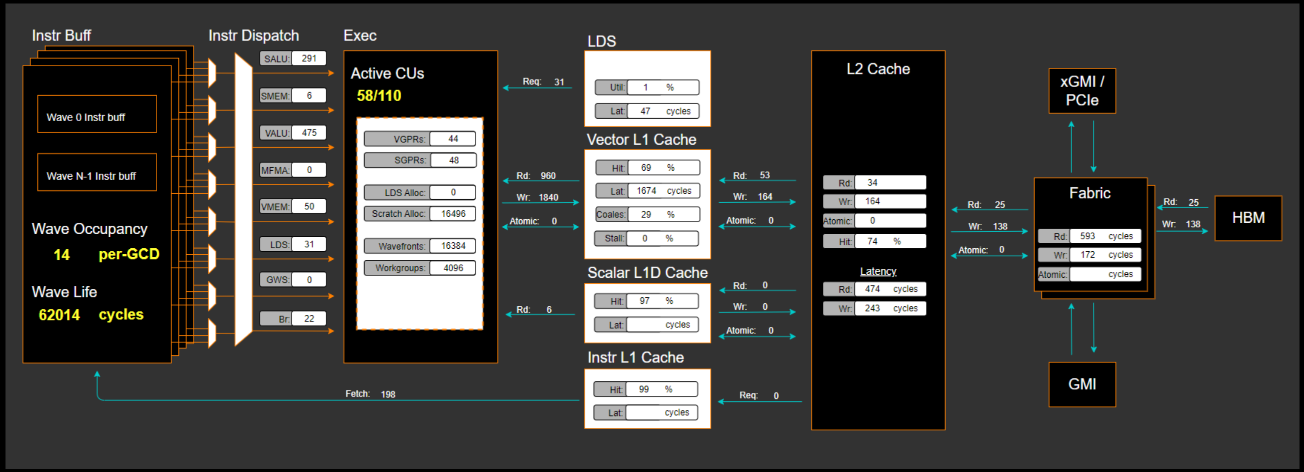
```
$ rocprof-compute profile -n vcopy_all -- ./vcopy 1048576 256
```

A new directory will be created called `workloads/vcopy_all`

Analyze the profiled workload:

```
$ rocprof-compute analyze -p workloads/vcopy_all/mi200/ --gui
```

Open web page http://IP:8050/

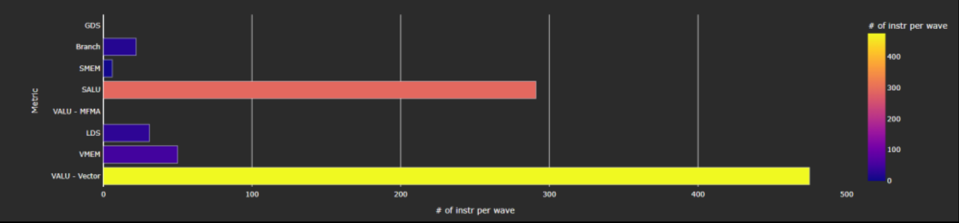


2. System Speed-of-Light

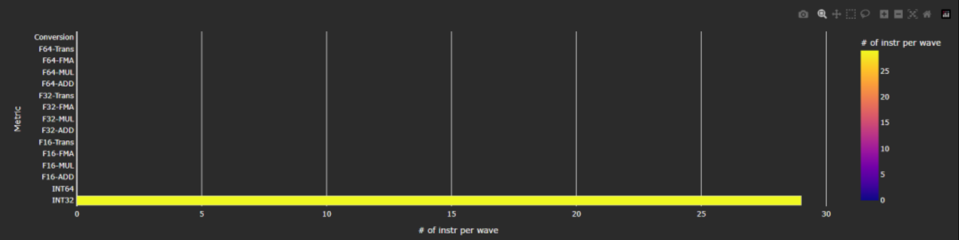
Metric	Value	Unit	Peak	PoP
VALU FLOPs	0.00	Gflop	23936.00	0.00
VALU IOPs	89.14	Giop	23936.00	0.37
MFMA FLOPs (BF16)	0.00	Gflop	95744.00	0.00
MFMA FLOPs (F16)	0.00	Gflop	191488.00	0.00
MFMA FLOPs (F32)	0.00	Gflop	47872.00	0.00
MFMA FLOPs (F64)	0.00	Gflop	47872.00	0.00
MFMA IOPs (Int8)	0.00	Giop	191488.00	0.00
Active CUs	58.00	Cus	110.00	52.73

10. Compute Units - Instruction Mix

10.1 Instruction Mix



10.2 VALU Arithmetic Instr Mix



Rocprof-compute analyze with Grafana™ GUI

We use the example sample/vcopy.cpp from the Rocprof-compute installation

```
$ wget https://raw.githubusercontent.com/ROCm/rocprofiler-compute/refs/heads/develop/sample/vcopy.cpp
```

Compile with hipcc:

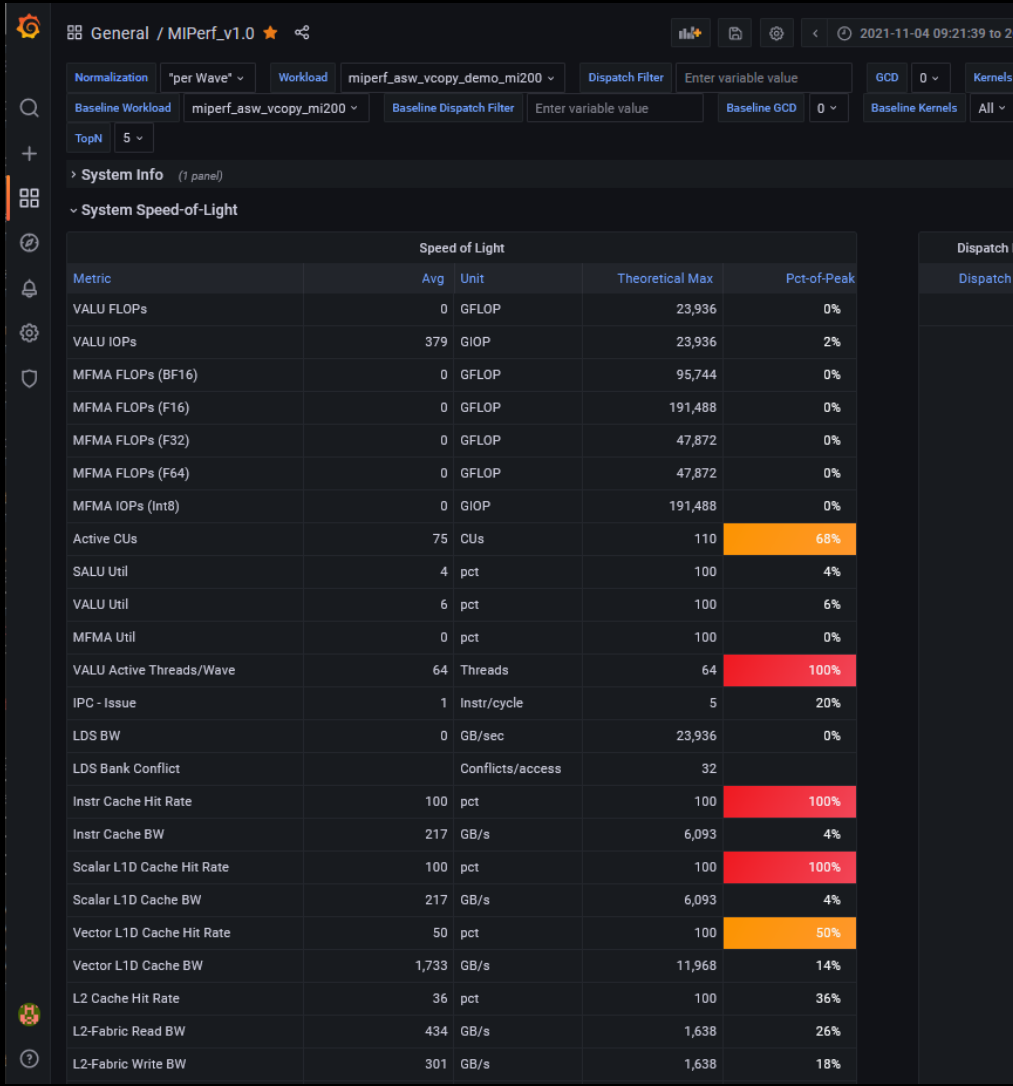
```
$ hipcc --offload-arch=gfx90a -o vcopy vcopy.cpp
```

Profile with Rocprof-compute:

```
$ rocprof-compute profile -n vcopy_all -- ./vcopy -n 1048576 -b 256
```

A new directory will be created called `workloads/vcopy_all`

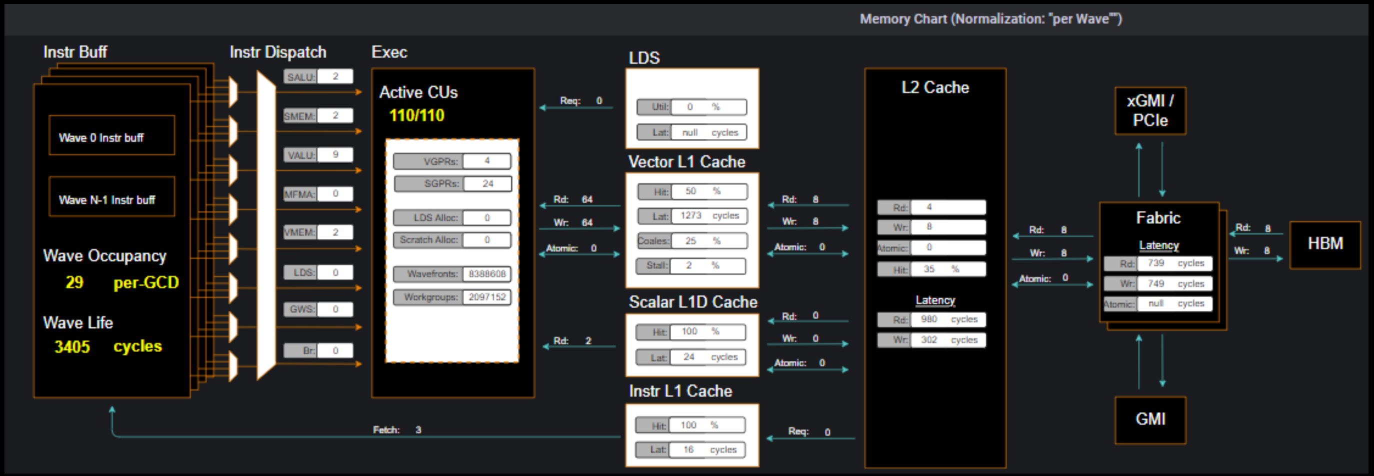
```
$ rocprof-compute database --import [connection options] -w workloads/vcopy_demo/mi200/
ROC Profiler:  /usr/bin/rocprof

--------
Import Profiling Results
--------

Pulling data from  /root/test/workloads/vcopy_demo/mi200
The directory exists
Found sysinfo file
KernelName shortening enabled
Kernel name verbose level: 2
Password:
Password recieved
-- Conversion & Upload in Progress –
… …
9 collections added.
Workload name uploaded
-- Complete! --
```



Key Insights from Rocprof-compute Analyzer

Initial assessment with kernel statistics

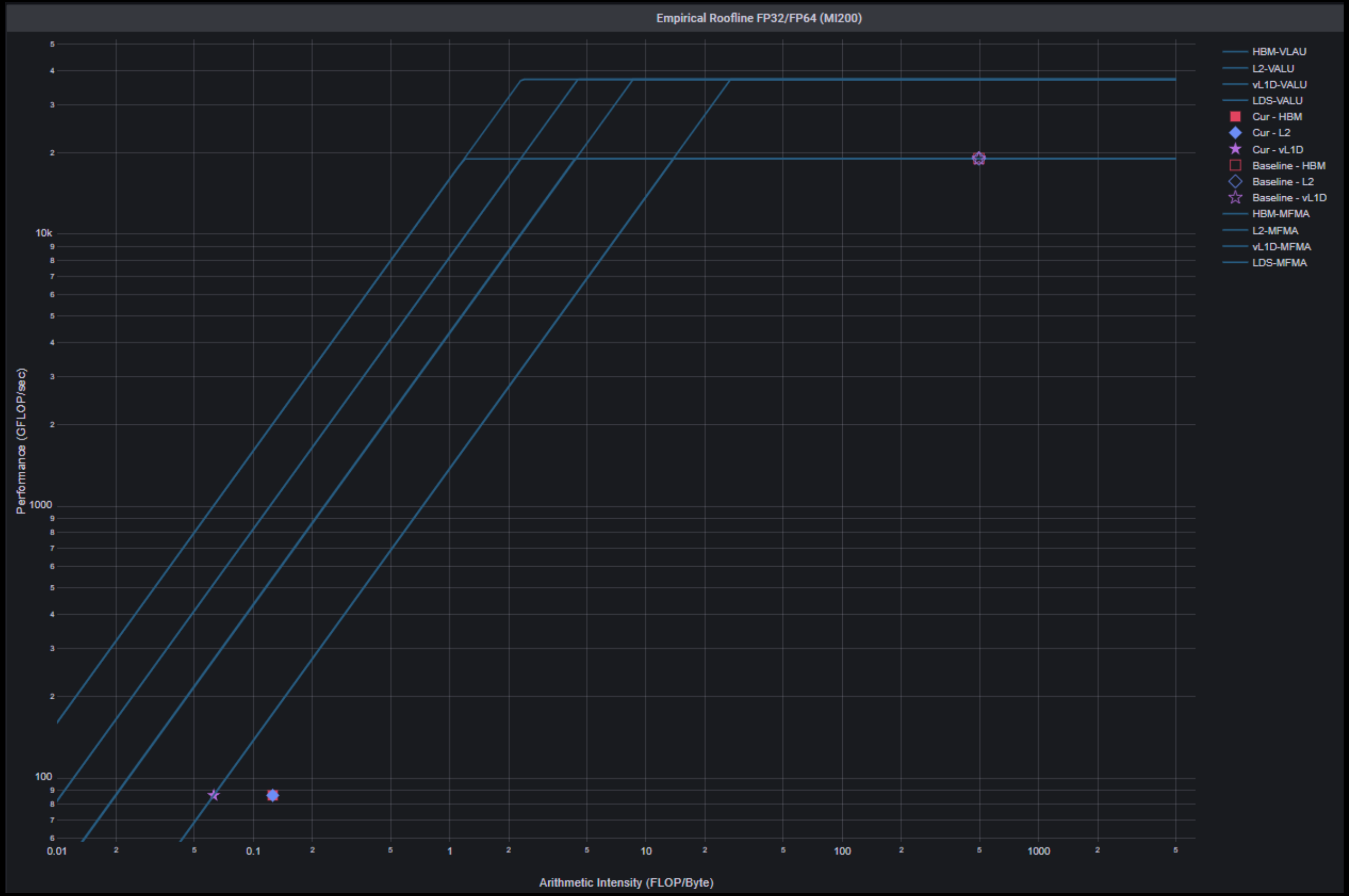
Initial Assessment	Instruction/data flow	Speed-of-Light (SOL)	
Rocprof-compute tooling support	System SOL	Memory Chart	Kernel statistics



Speed of Light				
Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
VALU FLOPs	0	GFLOP	23,936	0%
VALU IOPs	433	GIOP	23,936	2%
MFMA FLOPs (BF16)	0	GFLOP	95,744	0%
MFMA FLOPs (F16)	0	GFLOP	191,488	0%
MFMA FLOPs (F32)	0	GFLOP	47,872	0%
MFMA FLOPs (F64)	0	GFLOP	47,872	0%
MFMA IOPs (Int8)	0	GIOP	191,488	0%
Active CUs	110	CUs	110	100%
SALU Util	3	pct	100	3%
VALU Util	8	pct	100	8%
MFMA Util	0	pct	100	0%
VALU Active Threads/Wave	64	Threads	64	100%
IPC - Issue	1	Instr/cycle	5	20%
LDS BW	0	GB/sec	23,936	0%
LDS Bank Conflict		Conflicts/access	32	
Instr Cache Hit Rate	100	pct	100	100%

Roofline: the first-step characterization of workload performance

Workload characterization	Compute bound	Memory bound	L1/L2 cache access	Performance margin
Rocprof-compute tooling support	System SOL	Memory Chart	Kernel statistics	



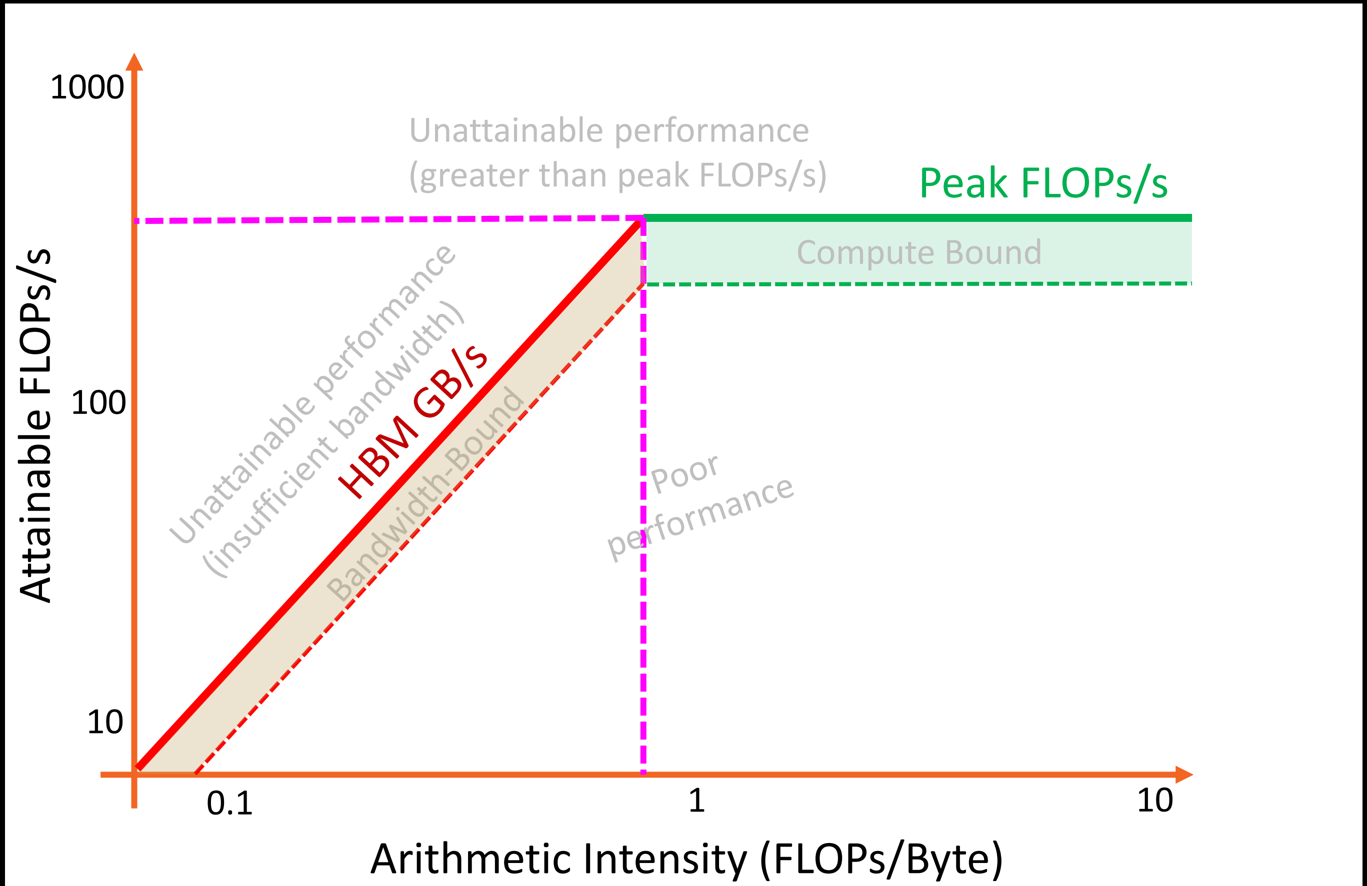
Top Kernels

Name	Calls	Performance	HBM BW	Total Duration	Avg Duration	AI (Vector L1D Cache	AI (L2 Cache)	AI (HBM)	Total FLOPs	VALU FLOPs	MFMA FLOPs (F16)	MFMA FLOPs (BF16)
void dot_kernel<doubl...	100	86.5 GFLOPS	689 GB/s	244 ms	2.44 ms	0.063	0.126	0.126	210,583,552	210,583,552	0	0
void triad_kernel<dou...	100	111 GFLOPS	1.33 TB/s	189 ms	1.89 ms	0.042	0.083	0.083	209,715,200	209,715,200	0	0
void add_kernel<doubl...	100	55.7 GFLOPS	1.34 TB/s	188 ms	1.88 ms	0.021	0.042	0.042	104,857,600	104,857,600	0	0
void copy_kernel<dou...	100	0 GFLOPS	1.37 TB/s	122 ms	1.22 ms	0	0	0	0	0	0	0
void mul_kernel<doubl...	100	86.1 GFLOPS	1.38 TB/s	122 ms	1.22 ms	0.031	0.063	0.063	104,857,600	104,857,600	0	0

Background - What is a roofline?

Background – What is Roofline

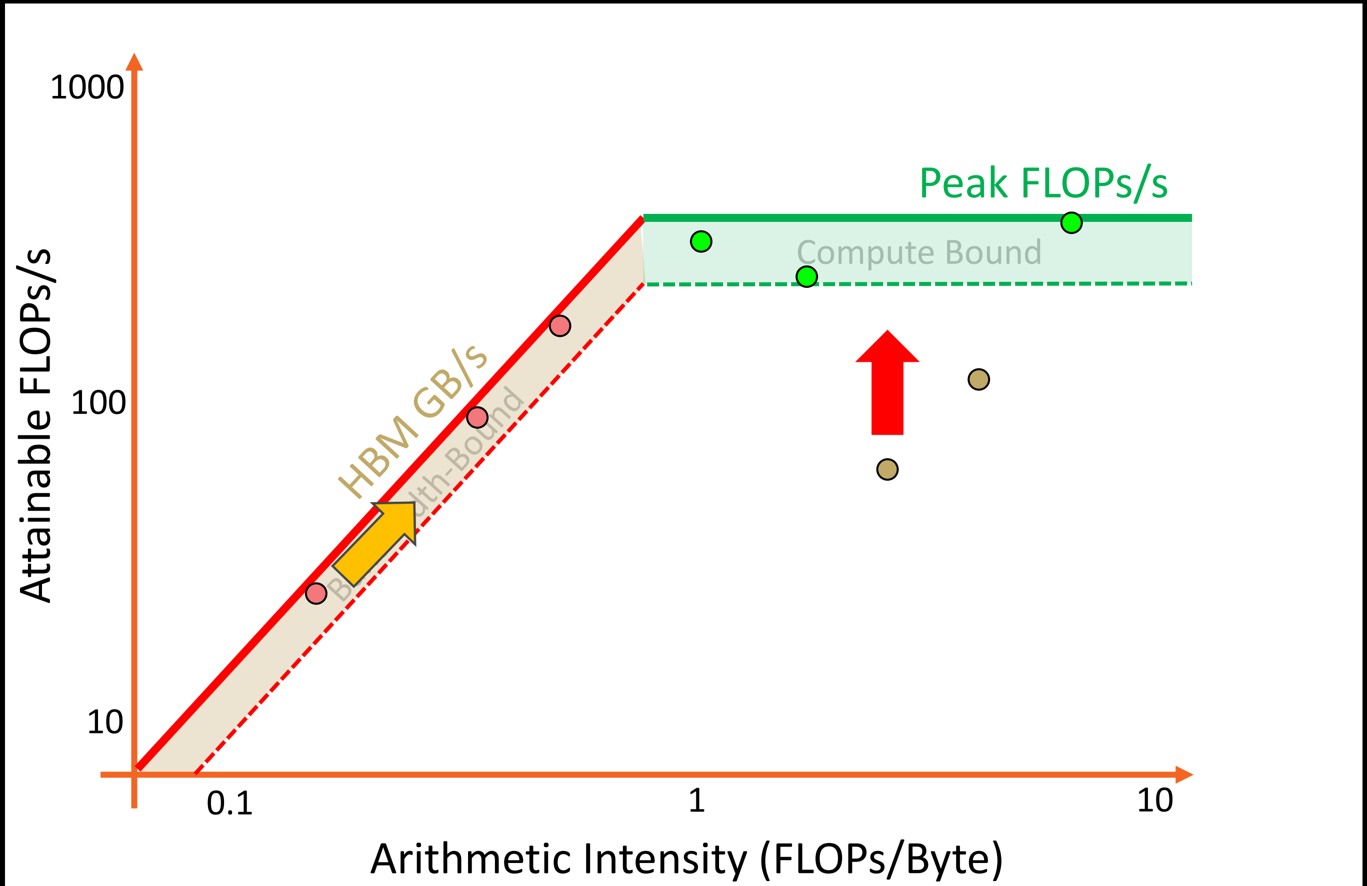
- Attainable FLOPs/s =
  - $min \begin{cases} Peak\ FLOPs/s \\ AI * Peak\ GB/s \end{cases}$
- Machine Balance:
  - Where $AI = \frac{Peak\ FLOPs/s}{Peak\ GB/s}$
- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound
  - Poor Performance



AMD
together we advance_

Background – What is "Good" Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities
  - Kernels near the roofline are making good use of computational resources
    - Kernels can have low performance (FLOPS/s), but make good use of BW
  - Increase arithmetic intensity when bandwidth limited
    - Reducing data movement increases AI
  - Kernels not near the roofline *should** have optimizations that can be made to get closer to the roofline

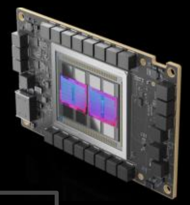


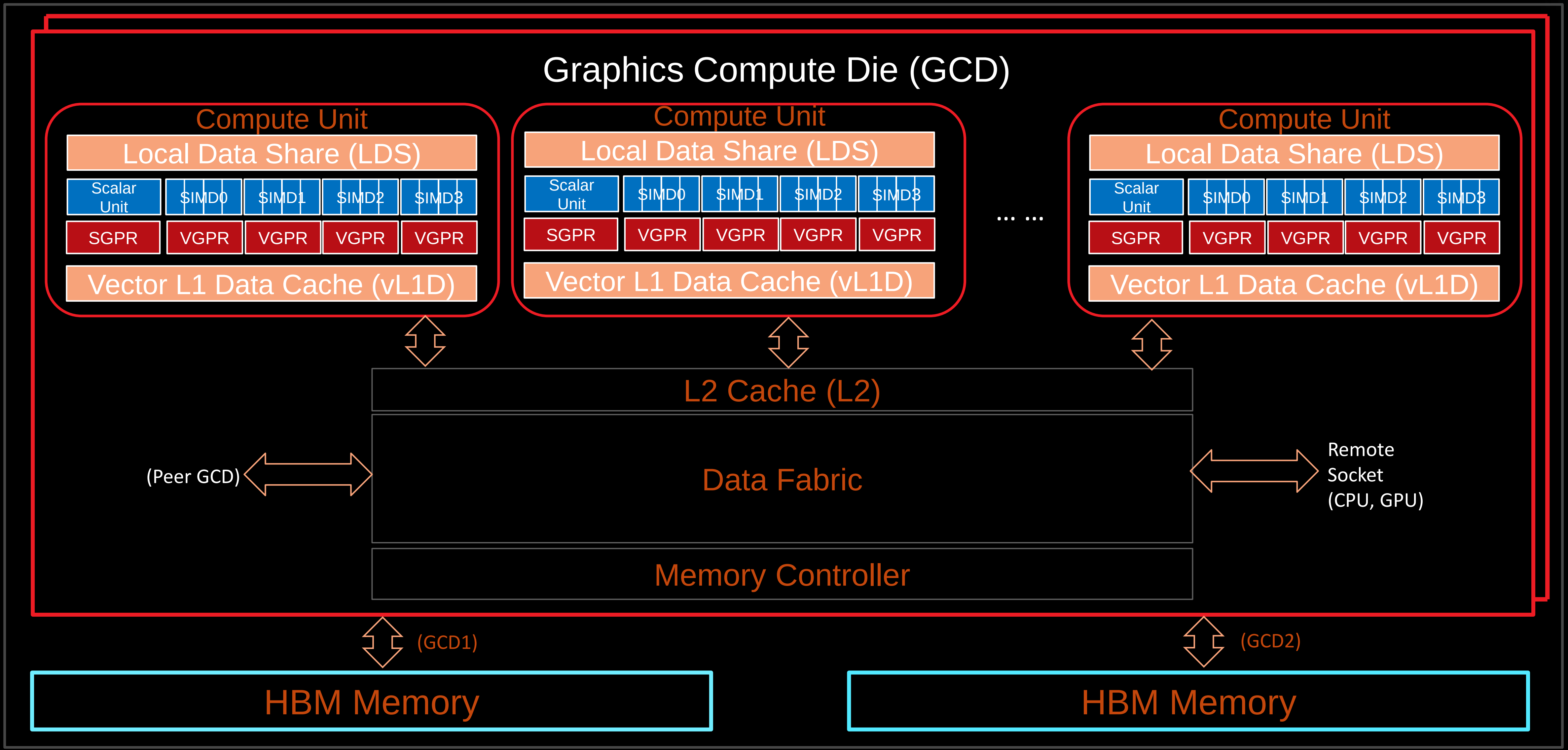
AMD
together we advance_

Roofline Calculations on AMD Instinct™ MI200 GPUs

[Public]

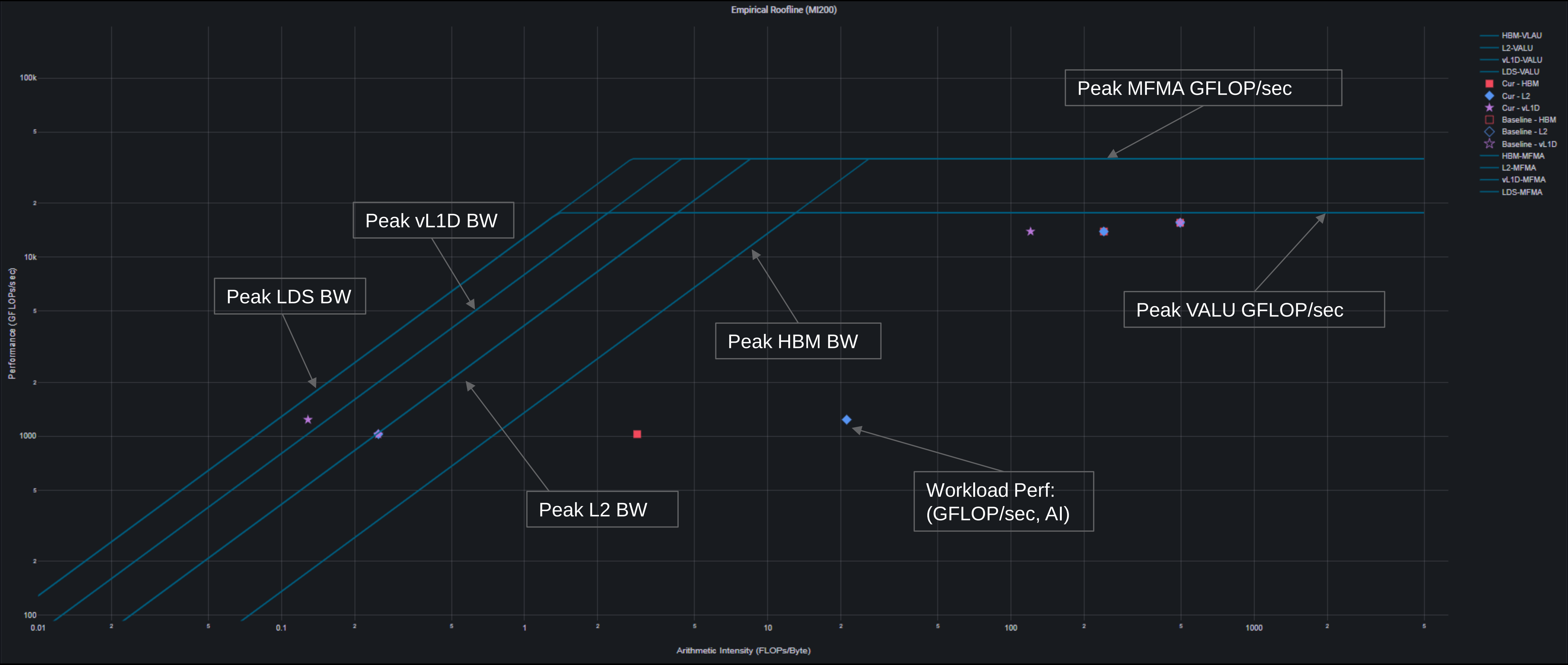
Overview - AMD Instinct™ MI200 Architecture





[Public]

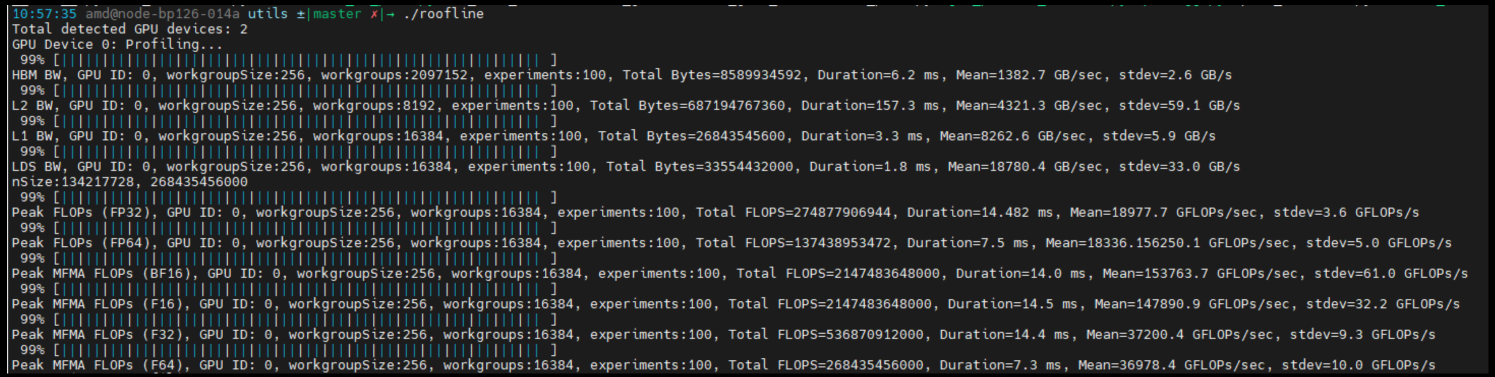
Empirical Hierarchical Roofline on MI200 - Overview



AMD
together we advance_

Empirical Hierarchical Roofline on MI200 – Roofline Benchmarking

- Empirical Roofline Benchmarking
  - Measure achievable Peak FLOPS
    - VALU: F32, F64
    - MFMA: F16, BF16, F32, F64
  - Measure achievable Peak BW
    - LDS
    - Vector L1D Cache
    - L2 Cache
    - HBM



- Internally developed micro benchmark algorithms
  - Peak VALU FLOP: axpy
  - Peak MFMA FLOP: Matrix multiplication based on MFMA intrinsic
  - Peak LDS/vL1D/L2 BW: Pointer chasing
  - Peak HBM BW: Streaming copy

AMD
together we advance_

Empirical Hierarchical Roofline on MI200 – Perfmon counters

- Weight
  - ADD: 1
  - MUL: 1
  - FMA: 2
  - Transcendental: 1
- FLOP Count
  - VALU: derived from VALU math instructions (assuming 64 active threads)
  - MFMA: count FLOP directly, in unit of 512
- Transcendental Instructions (7 in total)
  - e^x, $\log(x)$: F16, F32
  - $\frac{1}{x}$, $\sqrt{x}$, $\frac{1}{\sqrt{x}}$: F16, F32, F64
  - $\sin x$, $\cos x$: F16, F32
- Profiling Overhead
  - Require 3 application replays

```
v_rcp_f64_e32 v[4:5], v[2:3]
v_sin_f32_e32 v2, v2
v_cos_f32_e32 v2, v2
v_rsq_f64_e32 v[6:7], v[2:3]
v_sqrt_f32_e32 v3, v2
v_log_f32_e32 v2, v2
v_exp_f32_e32 v2, v2
```

ID	HW Counter	Category
1	SQ_INSTS_VALU_ADD_F16	FLOP counter
2	SQ_INSTS_VALU_MUL_F16	FLOP counter
3	SQ_INSTS_VALU_FMA_F16	FLOP counter
4	SQ_INSTS_VALU_TRANS_F16	FLOP counter
5	SQ_INSTS_VALU_ADD_F32	FLOP counter
6	SQ_INSTS_VALU_MUL_F32	FLOP counter
7	SQ_INSTS_VALU_FMA_F32	FLOP counter
8	SQ_INSTS_VALU_TRANS_F32	FLOP counter
9	SQ_INSTS_VALU_ADD_F64	FLOP counter
10	SQ_INSTS_VALU_MUL_F64	FLOP counter
11	SQ_INSTS_VALU_FMA_F64	FLOP counter
12	SQ_INSTS_VALU_TRANS_F64	FLOP counter
13	SQ_INSTS_VALU_INT32	IOP counter
14	SQ_INSTS_VALU_INT64	IOP counter
15	SQ_INSTS_VALU_MFMA_MOPS_I8	IOP counter

ID	HW Counter	Category
16	SQ_INSTS_VALU_MFMA_MOPS_F16	FLOP counter
17	SQ_INSTS_VALU_MFMA_MOPS_BF16	FLOP counter
18	SQ_INSTS_VALU_MFMA_MOPS_F32	FLOP counter
19	SQ_INSTS_VALU_MFMA_MOPS_F64	FLOP counter
20	SQ_LDS_IDX_ACTIVE	LDS Bandwidth
21	SQ_LDS_BANK_CONFLICT	LDS Bandwidth
22	TCP_TOTAL_CACHE_ACCESSES_sum	vL1D Bandwidth
23	TCP_TCC_WRITE_REQ_sum	L2 Bandwidth
24	TCP_TCC_ATOMIC_WITH_RET_REQ_sum	L2 Bandwidth
25	TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum	L2 Bandwidth
26	TCP_TCC_READ_REQ_sum	L2 Bandwidth
27	TCC_EA_RDREQ_sum	HBM Bandwidth
28	TCC_EA_RDREQ_32B_sum	HBM Bandwidth
29	TCC_EA_WRREQ_sum	HBM Bandwidth
3	TCC_EA_WRREQ_64B_sum	HBM

Empirical Hierarchical Roofline on MI200 - Arithmetic

Total_FLOP = 64 ∗ (SQ_INSTS_VALU_ADD_F16 + SQ_INSTS_VALU_MUL_F16 + SQ_INSTS_VALU_TRANS_F16 + 2 ∗ SQ_INSTS_VALU_FMA_F16)
+ 64 ∗ (SQ_INSTS_VALU_ADD_F32 + SQ_INSTS_VALU_MUL_F32 + SQ_INSTS_VALU_TRANS_F32 + 2 ∗ SQ_INSTS_VALU_FMA_F32)
+ 64 ∗ (SQ_INSTS_VALU_ADD_F64 + SQ_INSTS_VALU_MUL_F64 + SQ_INSTS_VALU_TRANS_F64 + 2 ∗ SQ_INSTS_VALU_FMA_F64)
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F16
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_BF16
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F32
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F64

Total_IOP = 64 ∗ (SQ_INSTS_VALU_INT32 + SQ_INSTS_VALU_INT64)

LDS_{BW} = 32 ∗ 4 ∗ (SQ_LDS_IDX_ACTIVE – SQ_LDS_BANK_CONFLICT)

$vL1D_{BW}$ = 64 ∗ TCP_TOTAL_CACHE_ACCESSES_sum

$L2_{BW}$ = 64 ∗ TCP_TCC_READ_REQ_sum
+ 64 * TCP_TCC_WRITE_REQ_sum
+ 64 * (TCP_TCC_ATOMIC_WITH_RET_REQ_sum + TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum)

HBM_{BW} = 32 * TCC_EA_RDREQ_32B_sum + 64 * (TCC_EA_RDREQ_sum - TCC_EA_RDREQ_32B_sum)
+ 32 * (TCC_EA_WRREQ_sum – TCC_EA_WRREQ_64B_sum) + 64 * TCC_EA_WRREQ_64B_sum

$$AI_{LDS} \quad \frac{TOTAL\_FLOP}{LDS_{BW}}$$

$$AI_{vL1D} \quad \frac{TOTAL\_FLOP}{vL1D_{BW}}$$

$$AI_{L2} \quad \frac{TOTAL\_FLOP}{L2_{BW}}$$

$$AI_{HBM} = \frac{TOTAL\_FLOP}{HBM_{BW}}$$

*** All calculations are subject to change***

AMD
together we advance_

Empirical Hierarchical Roofline on MI200 - Manual Rocprof

- For those who like getting their hands dirty
- Generate input file
  - See example roof-counters.txt →
- Run rocprof

```
foo@bar:~$ rocprof -i roof-counters.txt --timestamp on ./myCoolApp
```

- Analyze results
  - Load `results.csv` output file in csv viewer of choice
  - Derive final metric values using equations on previous slide
- Profiling Overhead
  - Requires one application replay for each pmc line

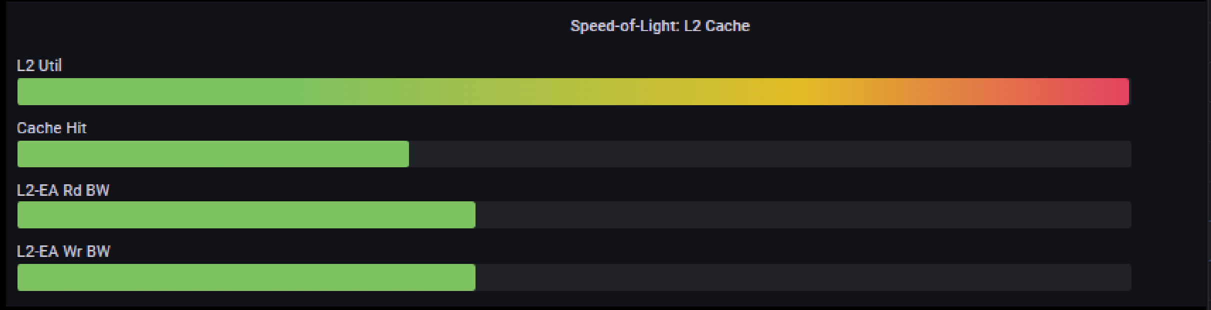
```
## roof-counters.txt

# FP32 FLOPs
pmc: SQ_INSTS_VALU_ADD_F32 SQ_INSTS_VALU_MUL_F32 SQ_INSTS_VALU_FMA_F32 SQ_INSTS_VALU_TRANS_F32

# HBM Bandwidth
pmc: TCC_EA_RDREQ_sum TCC_EA_RDREQ_32B_sum TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_sum

# LDS Bandwidth
pmc: SQ_LDS_IDX_ACTIVE SQ_LDS_BANK_CONFLICT

# L2 Bandwidth
pmc: TCP_TCC_READ_REQ_sum TCP_TCC_WRITE_REQ_sum TCP_TCC_ATOMIC_WITH_RET_REQ_sum
TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum

# vL1D Bandwidth
pmc: TCP_TOTAL_CACHE_ACCESSES_sum
```

Rocprof-compute Performance Analyzer (cont..)

Subsystem performance analysis

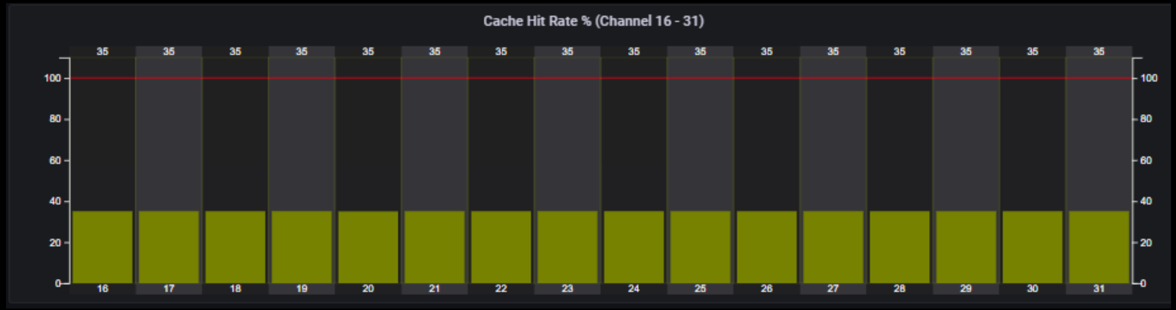
Memory subsystems
- L2 Cache
- HBM access
- LDS
- vL1D

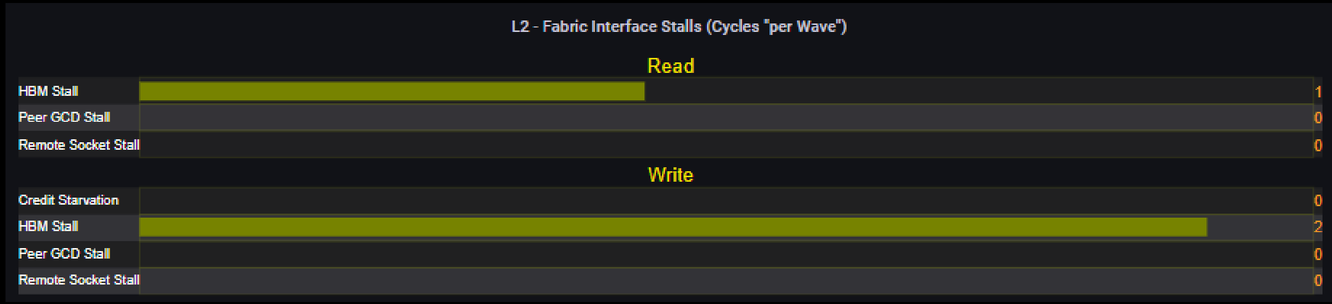
Rocprof-compute tooling support
- L2 Cache SOL
- L2 fabric metrics
- Per-channel statistics



L2 - Fabric Transactions

Metric	Avg	Min	Max	Unit
Read BW	693,148,700,953	664,565,016,054	695,197,543,698	Bytes per Sec
Write BW	692,659,558,092	664,096,634,666	694,705,946,653	Bytes per Sec
Read (32B)	0	0	0	Req per Sec
Read (Uncached 32...	2,304,240	1,434,649	2,370,898	Req per Sec
Read (64B)	10,830,448,452	10,383,828,376	10,862,461,620	Req per Sec
HBM Read	10,830,362,679	10,383,764,324	10,862,381,992	Req per Sec
Write (32B)	0	0	0	Req per Sec
Write (Uncached 32...	0	0	0	Req per Sec
Write (64B)	10,822,805,595	10,376,509,917	10,854,780,416	Req per Sec
HBM Write	10,822,801,389	10,376,488,102	10,854,762,613	Req per Sec
Read Latency	739	732	801	Cycles
Write Latency	749	737	784	Cycles
Atomic Latency				Cycles
Read Stall	3	2	3	pct
Write Stall	6	5	8	pct





AMD
together we advance_

Shader compute components

Shader compute | Wavefront life | Instruction mix | Floating/ Integer Ops | Compute pipeline

MFMA Arithmetic Instr Mix

MFMA Instr	Count
MFMA-I8	0
MFMA-F16	0
MFMA-BF16	0
MFMA-F32	0
MFMA-F64	995

Wavefront Runtime Stats

Metric	Avg	Min	Max	Unit
Kernel Time (Nanosec)	6,197,098	6,178,719	6,463,519	ns
Kernel Time (Cycles)	9,007,899	8,905,122	9,137,368	Cycle
Instr/wavefront	18	18	18	Instr/wavefro...
Wave Cycles	3,405	3,335	3,455	Cycles/wave
Dependency Wait Cycles	3,209	3,186	3,240	Cycles/wave
Issue Wait Cycles	165	112	193	Cycles/wave
Active Cycles	64	64	64	Cycles/wave
Wavefront Occupancy	3,198	3,166	3,210	Wavefronts

Rocprof-compute profile – Roofline only

Profile with roofline:

```
$ rocprof-compute profile -n roofline_case_app --roof-only -- <CMD> <ARGS>
```

Analyze the profiled workload:

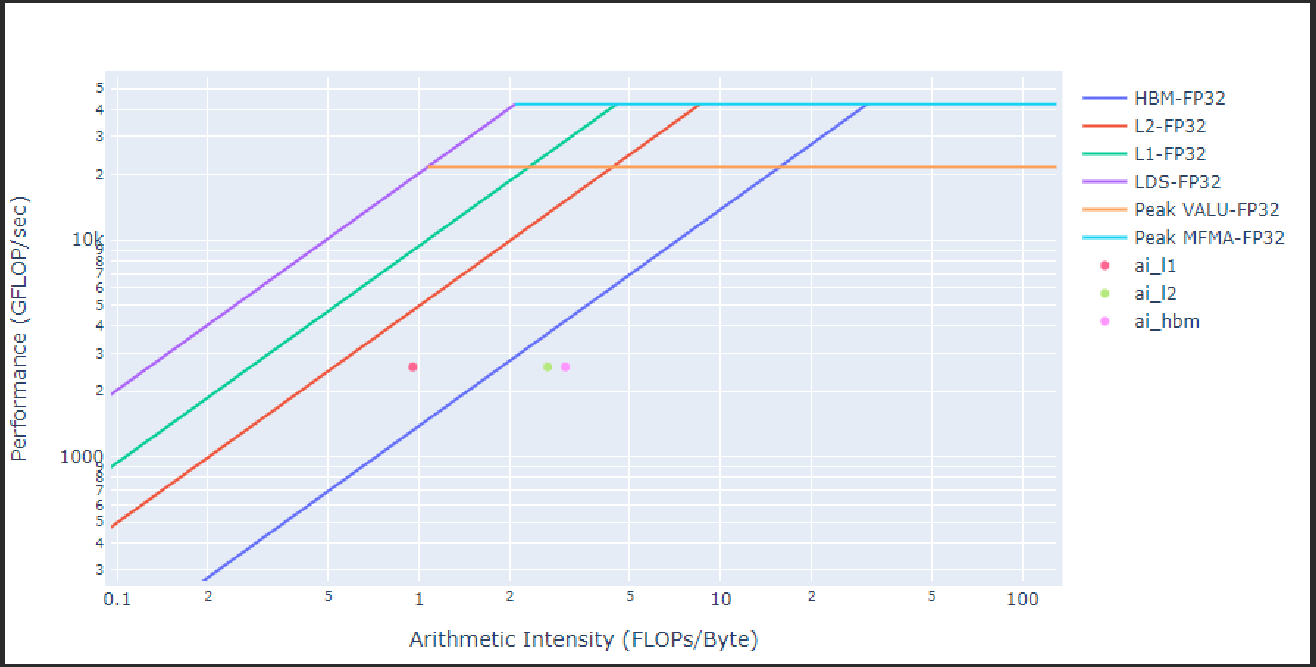
```
$ rocprof-compute analyze -p path/to/workloads/roofline_case_app/mi200 --gui
```

Open web page http://IP:8050/

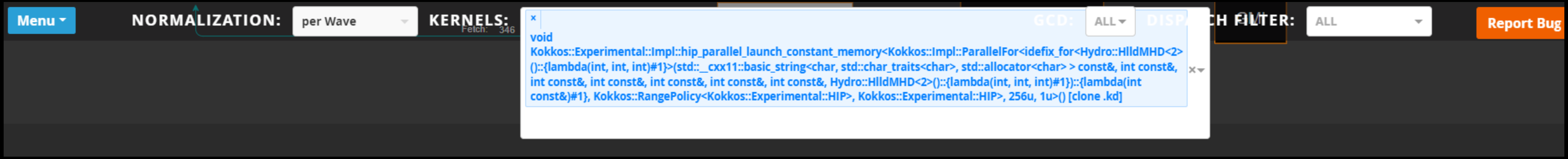
When profile with --roof-only, a PDF with the roofline will be created. In order to see the name of the kernels, add the --kernel-names and a second PDF will be created with names for the kernel markers:

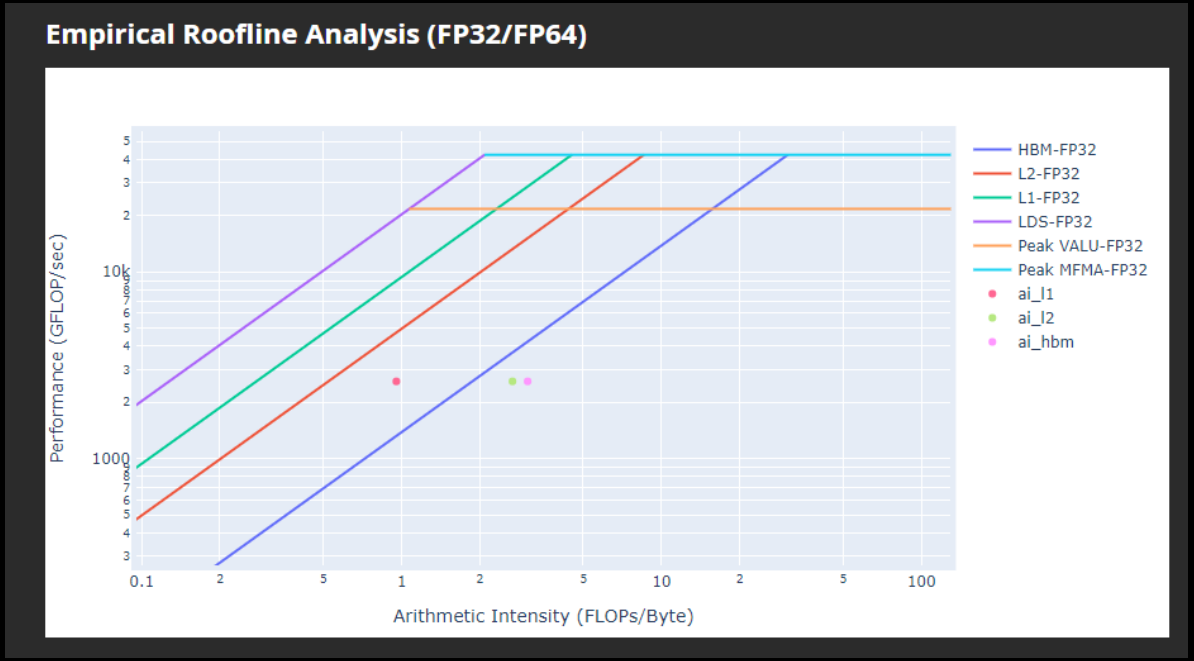
```
$ rocprof-compute profile -n roofline_case_app --roof-only --kernel-names -- <CMD> <ARGS>
```

Empirical Roofline Analysis (FP32/FP64)



Roofline Analysis – Kokkos code





- Roofline: the first-step characterization of workload performance
  - Workload characterization
    - Compute bound
    - Memory bound
    - Performance margin
    - L1/L2 cache accesses
- Thorough SoC perf analysis for each subsystem to identify bottlenecks
  - HBM
  - L1/L2
  - LDS
  - Shader compute
  - Wavefront dispatch
- Rocprof-compute tooling support
  - Roofline plot (float, integer)
  - Baseline roofline comparison
  - Kernel statistics

SPI Resource Allocation

- Dispatch Bound
  - Wavefront dispatching failure due to resources limitation
    - Wavefront slots
    - VGPR
    - SGPR
    - LDS allocation
    - Barriers
    - Etc.
  - Rocprof-compute tooling support
    - Shader Processor Input (SPI) metrics

6.2 SPI Resource Allocation

Metric	Avg	Min	Max	Unit
Wave request Failed (CS)	613303.00	613303.00	613303.00	Cycles
CS Stall	356961.00	356961.00	356961.00	Cycles
CS Stall Rate	62.95	62.95	62.95	Pct
Scratch Stall	0.00	0.00	0.00	Cycles
Insufficient SIMD Waveslots	0.00	0.00	0.00	Simd
Insufficient SIMD VGPRs	16252333.00	16252333.00	16252333.00	Simd
Insufficient SIMD SGPRs	0.00	0.00	0.00	Simd
Insufficient CU LDS	0.00	0.00	0.00	Cu
Insufficient CU Barries	0.00	0.00	0.00	Cu
Insufficient Bulky Resource	0.00	0.00	0.00	Cu
Reach CU Threadgroups Limit	0.00	0.00	0.00	Cycles
Reach CU Wave Limit	0.00	0.00	0.00	Cycles
VGPR Writes	4.00	4.00	4.00	Cycles/wave
SGPR Writes	5.00	5.00	5.00	Cycles/wave

What if Grafana and web GUI crashes when loading performance data? (real case)

When profiling produces too large data…

- We had an application that the realistic case was dispatching 6.7 million calls to kernels
- Executing Rocprof-compute without any options, it would take up to 36 hours to finish while single non instrumented execution takes less than 1 hour.
- HW counters add overhead
- We had totally around 9 GB of profiling data from 1 MPI process
- Uploading the data to a Grafana server was crashing Grafana server and we had to reboot the service
- Using standalone GUI was never finishing loading the data

- Rocprof-compute profile has an option called –k where you define which specific kernel to profile. You can define the id 0-9 of the top 10 kernels.
- This creates profiling data **only** for the selected kernel
- This way you can split the profiling data to 10 executions, one per kernel:
  - You can use different resources to do the experiments in parallel (remember there can be performance variation between different GPUs)
  - You can visualize each kernel

Profile with roofline for a specific kernel:

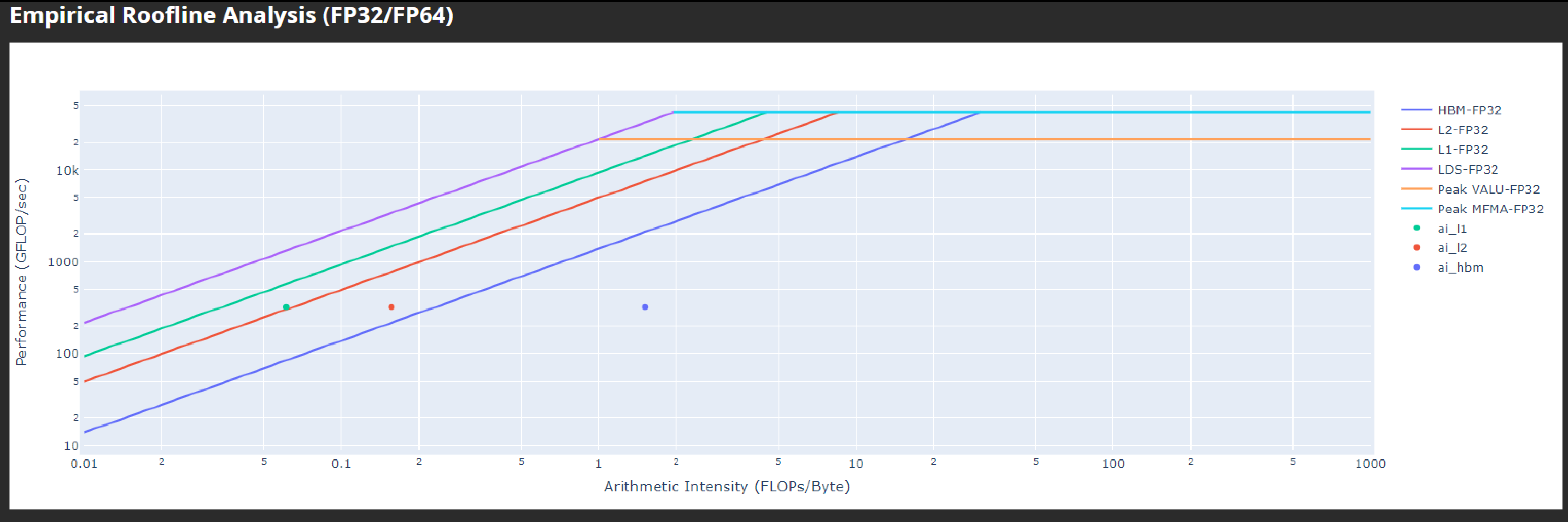
```
 $ srun -N 1 -n 1 --ntasks-per-node=1 --gpus=1 --hint=nomultithread rocprof-compute profile -n kernel_roof -k kernel_name --roof-only  -- ./binary args
```

Example – DAXPY with a loop in the kernel

DAXPY – with a loop in the kernel

```
#include <hip/hip_runtime.h>

__constant__ double a = 1.0f;

__global__
void daxpy (int n, double const* x, int incx, double* y, int incy)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
       for(int ll=0;ll<20;ll++) {
        y[i] = a*x[i] + y[i];
       }
}

int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

    double* d_x;
    double *d_y;
    hipMalloc(&d_x, size);
    hipMalloc(&d_y, size);

    int num_groups = (n+255)/256;
    int group_size = 256;
    daxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
    hipDeviceSynchronize();
```

Roofline



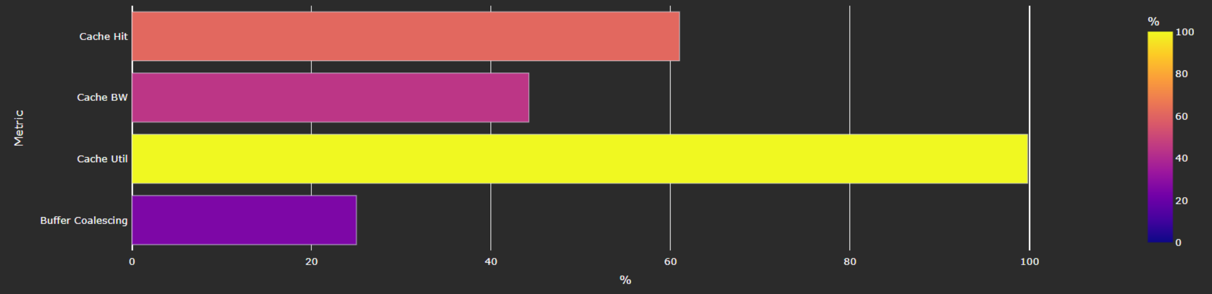
- Performance: almost 330 GFLOPs

Kernel execution time and L1D Cache Accesses

KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
daxpy(int, double const*, int, double*, int) [clone .kd]	1.00	2024491.00	2024491.00	2024491.00	100.00

16. Vector L1 Data Cache

16.1 Speed-of-Light



16.2 L1D Cache Stalls

Metric	Mean	Min	Max	unit
Stalled on L2 Data	73.69	73.69	73.69	Pct
Stalled on L2 Req	19.47	19.47	19.47	Pct
Tag RAM Stall (Read)	0.00	0.00	0.00	Pct
Tag RAM Stall (Write)	0.00	0.00	0.00	Pct
Tag RAM Stall (Atomic)	0.00	0.00	0.00	Pct

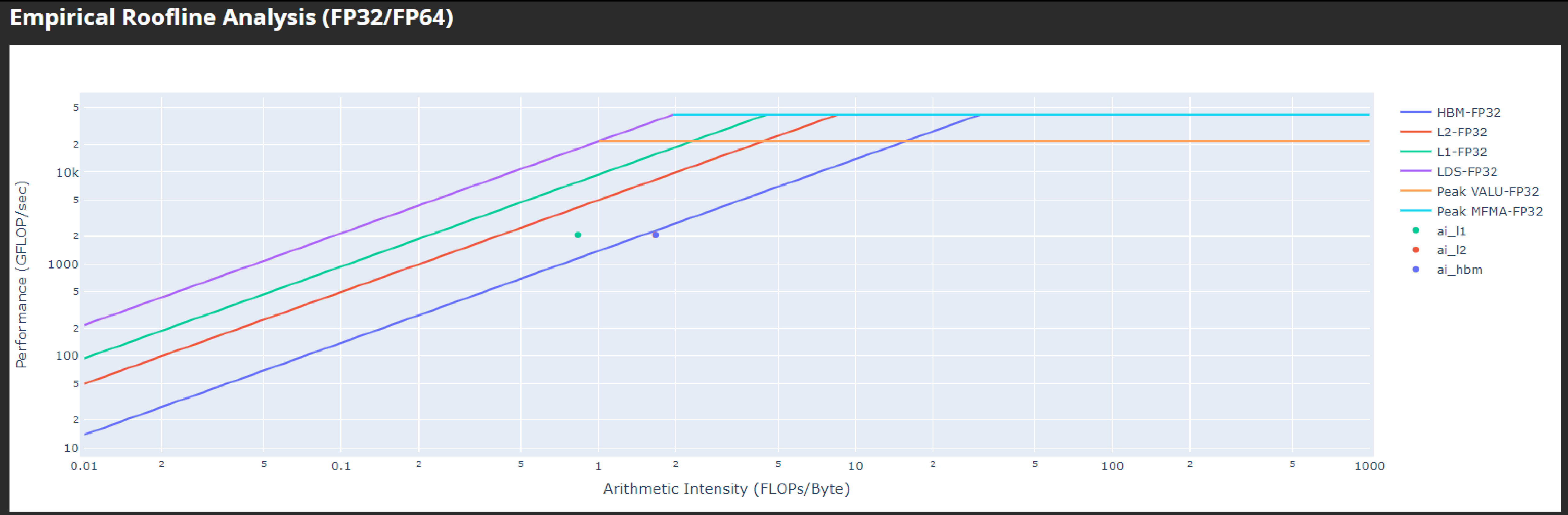
16.3 L1D Cache Accesses

Metric	Avg	Min	Max	Unit
Total Req	2624.00	2624.00	2624.00	Req per wave
Read Req	1344.00	1344.00	1344.00	Req per wave
Write Req	1280.00	1280.00	1280.00	Req per wave
Atomic Req	0.00	0.00	0.00	Req per wave
Cache BW	5291.66	5291.66	5291.66	Gb/s
Cache Accesses	656.00	656.00	656.00	Req per wave
Cache Hits	400.16	400.16	400.16	Req per wave
Cache Hit Rate	61.00	61.00	61.00	Pct

DAXPY – with a loop in the kernel - Optimized

```
#include <hip/hip_runtime.h>

__constant__ double a = 1.0f;

__global__
void daxpy (int n, double const* __restrict__ x, int incx, double* __restrict__ y, int incy)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
       for(int ll=0;ll<20;ll++) {
        y[i] = a*x[i] + y[i];
       }
}

int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

    double* d_x;
    double *d_y;
    hipMalloc(&d_x, size);
    hipMalloc(&d_y, size);

    int num_groups = (n+255)/256;
    int group_size = 256;
    daxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
    hipDeviceSynchronize();
```

Roofline - Optimized



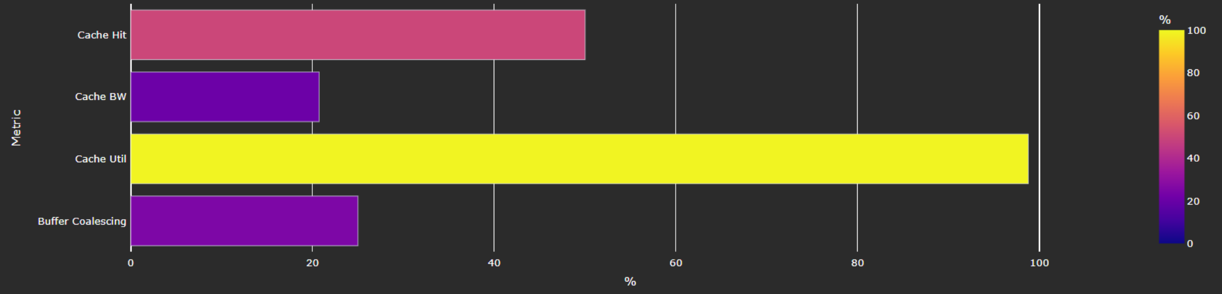
- Performance: almost 2 TFLOPs

Kernel execution time and L1D Cache Accesses - Optimized

KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
daxpy(int, double const*, int, double*, int) [clone .kd]	1.00	323522.00	323522.00	323522.00	100.00

6.2 times faster!

16.1 Speed-of-Light



16.2 L1D Cache Stalls

Metric	Mean	Min	Max	unit
Stalled on L2 Data	79.08	79.08	79.08	Pct
Stalled on L2 Req	15.17	15.17	15.17	Pct
Tag RAM Stall (Read)	0.00	0.00	0.00	Pct
Tag RAM Stall (Write)	0.00	0.00	0.00	Pct
Tag RAM Stall (Atomic)	0.00	0.00	0.00	Pct

16.3 L1D Cache Accesses

Metric	Avg	Min	Max	Unit
Total Req	192.00	192.00	192.00	Req per wave
Read Req	128.00	128.00	128.00	Req per wave
Write Req	64.00	64.00	64.00	Req per wave
Atomic Req	0.00	0.00	0.00	Req per wave
Cache BW	2480.60	2480.60	2480.60	Gb/s
Cache Accesses	48.00	48.00	48.00	Req per wave
Cache Hits	24.00	24.00	24.00	Req per wave
Cache Hit Rate	50.00	50.00	50.00	Pct
Invalidate	0.00	0.00	0.00	Req per wave

Guided Exercises

1. Launch Parameters
2. LDS Occupancy Limiter
3. VGPR Occupancy Limiter
4. Strided Data Access Pattern
5. Algorithmic Optimizations

Guided exercises: Logistics/Preamble

```
git clone https://github.com/amd/HPCTrainingExamples.git
cd HPCTrainingExamples/OmniperfExamples
```

- Feel free to clone the above repo and start working through the exercises
  - The READMEs are comprehensive walkthroughs on their own, I'll provide highlights in the talk

- To generate the output for these slides used Rocprofiler-compute from ROCm 6.4.0
  - As of ROCm 6.2.0, Omniperf was packaged with ROCm as an officially supported tool
  - In ROCm 6.3.0, Omniperf has been renamed to `rocprof-compute`
  - This is a module available to you on the training environment: `module load rocprofiler-compute/6.4.0`

- The numbers shown in the READMEs were generated using MI210 and MI300A accelerators, and the accelerator used is made clear in each case

- WARNING: For educational purposes implementations in these exercises are not fully-optimized kernels

Guided Exercises: Representative Optimization Tasks

- The Exercises are roughly in order of ease of development effort and performance impact:
  - Exercise 1: Verify Reasonable Launch Parameters
  - Exercise 2: Attempt to Cache Data in Shared Memory
  - Exercise 3: Determining a Source of Unexpected Resource Usage
  - Exercise 4: Verifying Efficient Data Access Patterns
  - Exercise 5: Analyzing an Algorithmic Change

- The underlying code is kept simple to emphasize the optimization techniques

- These slides are intended as a “Cheat Sheet” starting point providing:
  - Rocprof-compute commands to filter through output for common optimization concerns
  - Some optimization direction given certain Rocprof-compute output

Guided Exercises: Optimizing a yAx Kernel

- We'll be looking at a relatively simple kernel that solves the same problem in each exercise, yAx
  - yAx is a vector-matrix-vector product that can be implemented in serial as:

```
double result = 0.0;
for (int i = 0; i < n; i++){
  double temp = 0.0;
  for (int j = 0; j < m; j++){
    temp += A[i*m + j] * x[j];
  }
  result += y[i] * temp;
}
```

- Where:
  - A is a 1-D array of size n*m
  - x is an array of size m
  - y is an array of size n

Exercise 1: First things first, generate a roofline

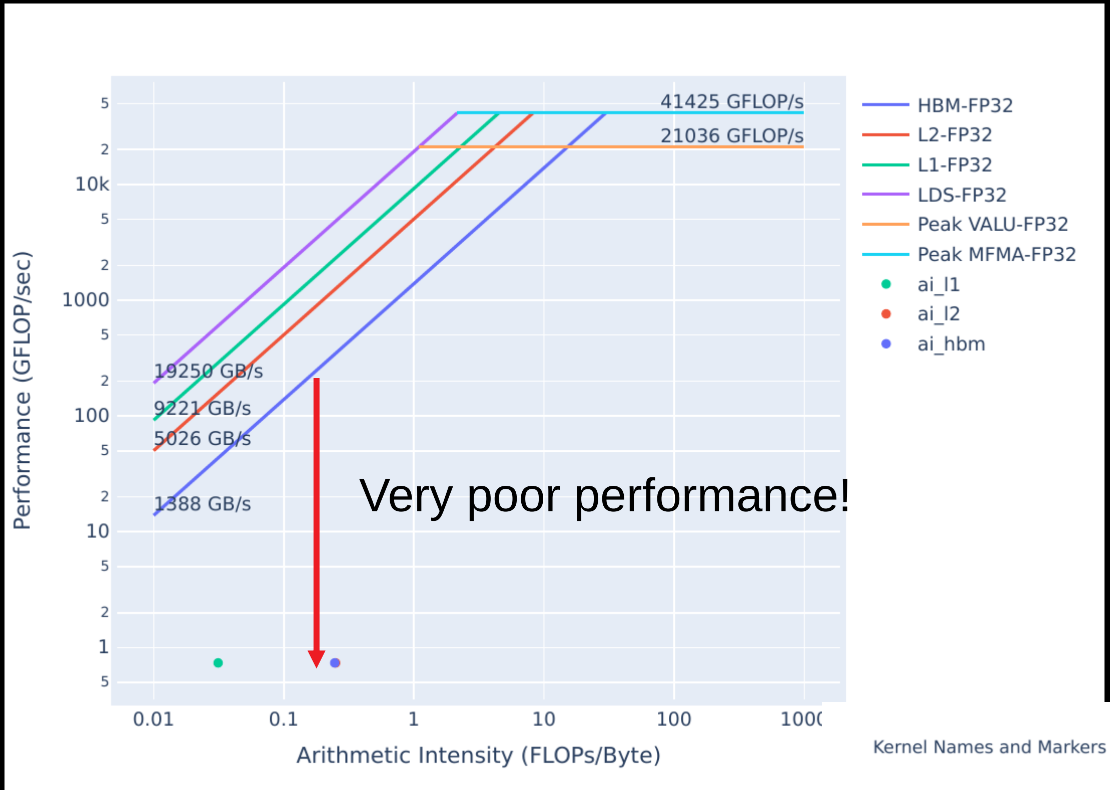
- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `rocprof-compute profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/MI300_A1` folder.
    - `--roof-only` generates PDF roofline plots, and does **not** generate any non-roofline profiling data
    - `--kernel-names` generates a separate PDF showing which kernel names correspond to which icons in the roofline

- Rooflines are a useful tool in determining which kernels are good optimization targets
  - Only one perspective of performance, kernel runtime cannot be inferred from the roofline

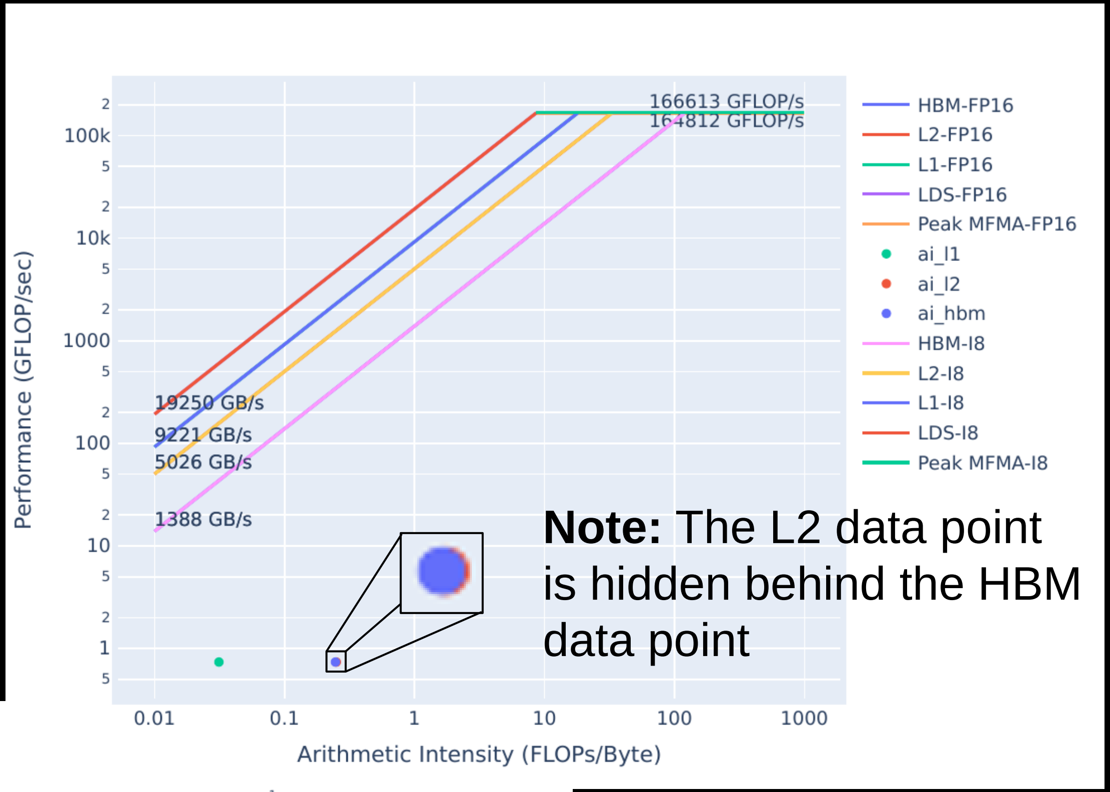
- Generated PDF roofline plots can have overlapping data points but should still be instructive
  - There are fixes to this, but they may be difficult to setup for different cluster installations
  - Generating the PDF plots from the command line interface should always work

- Complete sets of roofline plots and commands can be found in the READMEs for each exercise

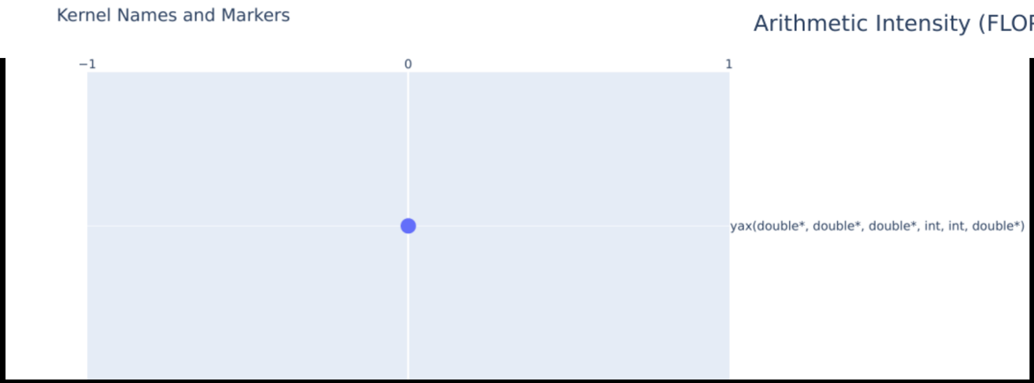
Exercise 1: Roofline plots



Very poor performance!



Note: The L2 data point is hidden behind the HBM data point



Kernel legend in a separate PDF

Exercise 1: Prep to find kernel launch parameters

- Launch parameters are given at the time of the kernel launch, as in lines 49 and 54:
  - `yax<<<grid,block>>>(y,A,x,n,m,result);`
    - Where grid and block are the kernel `yax`'s launch parameters
  - In problem, `grid = (4,1,1)`, and `block = (64,1,1)`
  - In solution, `grid = (2048,1,1)`, and `block = (64,1,1)`

- Sometimes launch parameters can be obfuscated by OpenMP® and other parallelism layers

- Rocprof-compute can easily show launch parameter information regardless of the code
  - You just need the dispatch ID – other forms of filtering may report aggregate launch parameters

- To generate profiling data, use the commands:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`
    - `--no-roof` saves time by not generating roofline data – profile commands can take a while

- Real benchmarks can take prohibitively long – use smaller representative problems when possible

Exercise 1: CLI Rocprof-compute comparisons are easy

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

Using problem as the baseline, and solution as the comparative

```
INFO Analysis mode = cli
  INFO [analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	543201153.00	9589864.0 (-98.23%)	543201153.00	9589864.0 (-98.23%)	543201153.00	9589864.0 (-98.23%)	100.00	100.0 (0.0%

56.6x speedup

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

Typically, difficult to pre-determine optimal launch parameters, so some experimentation often necessary

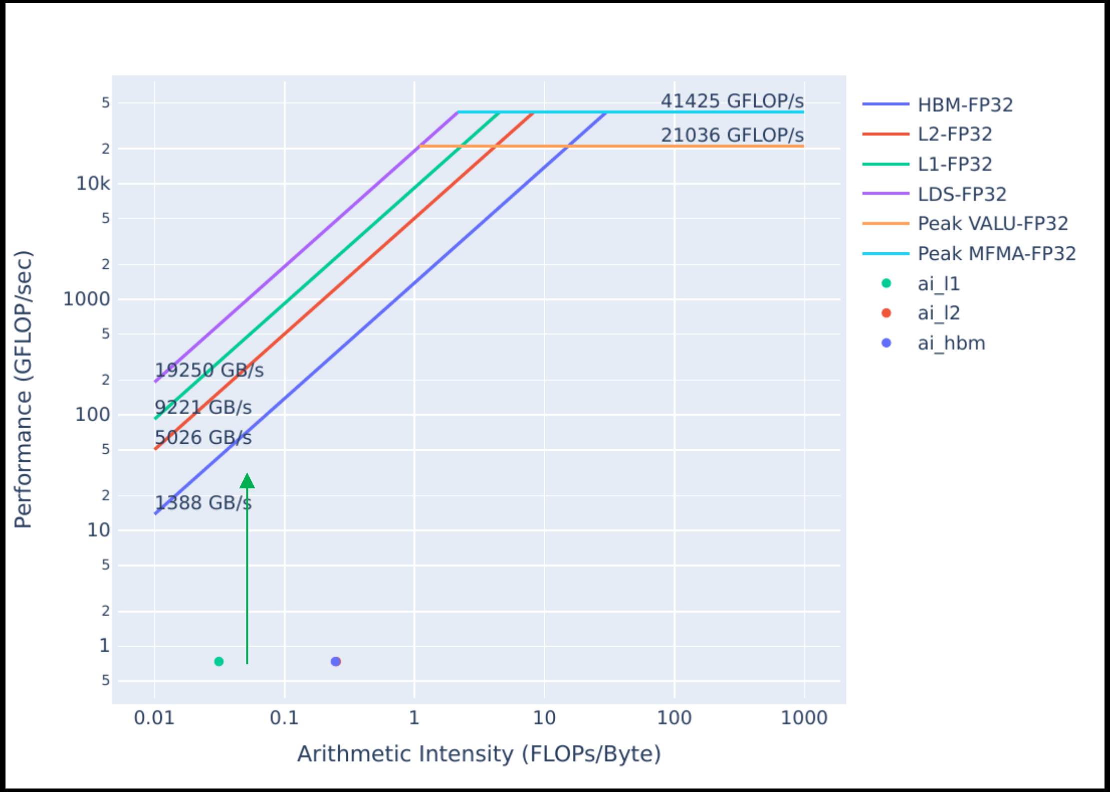
```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Metric_ID │ Max	Metric │ Max	Avg │ Unit	Avg │	Abs Diff	Min	Min			
7.1.0	Grid Size	256.00	131072.0 (51100.0%)	130816.00	256.00	131072.0 (51100.0%)	256.00	131072.0 (51100.0%)	Work items
7.1.1	Workgroup Size	64.00	64.0 (0.0%)	0.00	64.00	64.0 (0.0%)	64.00	64.0 (0.0%)	Work items
7.1.2	Total Wavefronts	4.00	2048.0 (51100.0%)	2044.00	4.00	2048.0 (51100.0%)	4.00	2048.0 (51100.0%)	Wavefronts

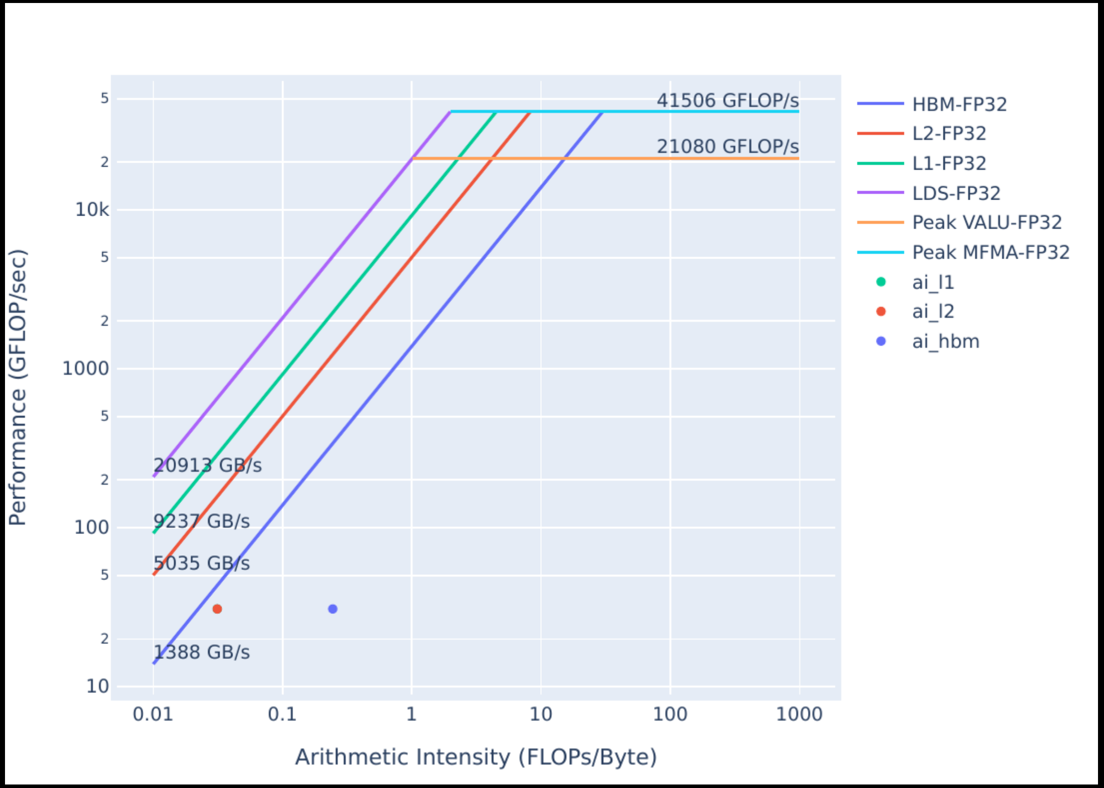
Increased launched wavefronts, which increases Grid Size

Exercise 1: Comparing problem and solution roofline plots

Problem FP32 Roofline Plot



Solution FP32 Roofline Plot



Generally, moving up and to the right is good

Exercise 1: It’s easy to check launch parameters

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

- `--block` filters the output to only show launch parameters

- Good launch parameters essential to a performant GPU kernel
  - Determining which parameters give the best performance usually requires experimenting

- Can be difficult to track down where launch parameters are set in code (OpenMP® may decide)

Exercise 2: Diagnosing a Shared Memory Occupancy Limiter

- Using LDS (Local Data Store – Shared Memory) to cache re-used data can be an effective optimization strategy

- Using **too much** LDS can restrict occupancy however, and reduce performance

- Line 12 in problem.cpp shows the allocation of LDS:
  - `__shared__ double tmp[fully_allocate_lds];`

- There are two solutions:
  - `solution-no-lds` removes the LDS allocation, and thus the occupancy limiter
  - `solution` reduces the size of the LDS allocation, removes occupancy limiter, and is faster than `solution-no-lds`
    - This is the solution used to generate the Rocprof-compute output in the next slide

- Rocprof-compute makes it easy to determine if LDS allocations restrict occupancy, as before profile with:
  - `rocprof-compute profile –n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile –n solution --no-roof -- ./solution.exe`

Exercise 2: LDS occupancy limiter – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.7
```

```
INFO Analysis mode = cli
  INFO [analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	7225180.00	5736816.0 (-20.6%)	7225180.00	5736816.0 (-20.6%)	7225180.00	5736816.0 (-20.6%)	100.00	100.0 (0.0%)

1.26x speedup

```
0.2 Dispatch List
```

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

```
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Peak	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	175.66	418.68 (138.35%)	3.33	Wavefronts	7296.00	7296.0 (0.0%)	2.41	5.74 (138.31%)

+ ~3% Occupancy (overall)

```
--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.7	Insufficient CU LDS	57.33	0.0 (-100.0%)	-57.33	57.33	0.0 (-100.0%)	57.33	0.0 (-100.0%)	Pct

Sharp decrease in Workgroup Manager stat

AMD
together we advance_

Exercise 2: Use SPI stats to determine if LDS limits occupancy

- Occupancy limiters can negatively impact performance
  - Occupancy increases don't always correspond to increased performance

- Workgroup Manager (SPI – Shader Processor Input) stats in Rocprofiler-compute indicate whether a kernel resource limits occupancy

- You can get the Workgroup Manager stat for LDS for a single kernel with dispatch ID 1:
  - `rocprof-compute analyze -p workloads/problem/MI300_A1 --dispatch 1 --block 2.1.15 6.2.7`

Note:

- In Rocprofiler-compute, the Workgroup Manager "insufficient resource" stats are percentages, meaning:
  - The magnitude of these fields **does not** necessarily indicate how severely occupancy is impacted
    - Changes to the Workgroup Manager stat do not directly translate to changes to overall occupancy, necessarily
  - If two fields are nonzero, the larger number indicates that resource is limiting occupancy more

Exercise 3: Diagnosing a register occupancy limiter

- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - The solution simply removes the assert
  - Admittedly the occupancy limit is very minor, but this is a good excuse to look at register usage.

- The types of registers on AMD GPUs are:
  - VGPRs (Vector General Purpose Registers): registers that can hold distinct values for each thread in the wavefront
  - SGPRs (Scalar General Purpose Registers): uniform across a wavefront. If possible, using these is preferable
  - AGPRs (Accumulation vector General Purpose Registers): special-purpose registers for MFMA (Matrix Fused Multiply-Add) operations, or low-cost register spills

- Using too many of one of these register types can impact occupancy and negatively impact performance

- We use the same profile commands to get the profiling data:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`

Exercise 3: Register occupancy limiter – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9993665.00	9666265.0 (-3.28%)	9993665.00	9666265.0 (-3.28%)	9993665.00	9666265.0 (-3.28%)	100.00	100.0 (0.0%

Minor speedup

```
0.2 Dispatch List
<omitted>
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Peak	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	430.98	427.36 (-0.84%)	-0.05	Wavefronts	7296.00	7296.0 (0.0%)	5.91	5.86 (-0.85%)

Similar occupancies

```
--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.5	Insufficient SIMD VGPRs	0.06	0.0 (-99.7%)	-0.06	0.06	0.0 (-99.7%)	0.06	0.0 (-99.7%)	Pct

Minor change in Workgroup Manager stat

```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Exact values might be slightly different, but conclusion stay the same

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
7.1.5	VGPRs	92.00	32.0 (-65.22%)	-60.00	92.00	32.0 (-65.22%)	92.00	32.0 (-65.22%)	Registers
7.1.6	AGPRs	132.00	0.0 (-100.0%)	-132.00	132.00	0.0 (-100.0%)	132.00	0.0 (-100.0%)	Registers
7.1.7	SGPRs	48.00	112.0 (133.33%)	64.00	48.00	112.0 (133.33%)	48.00	112.0 (133.33%)	Registers

Fewer VGPRs
No AGPRs
More SGPRs

Exercise 3: Register occupancy limiter – takeaways

- Seemingly innocuous function calls inside kernels **can** lead to unexpected performance characteristics
  - Asserts, and even excessive use of math functions in kernels can degrade performance
  - Can be difficult to construct clear examples of this, anecdotally

- In this case the occupancy limit is very minor

- AGPR usage in the absence of MFMA instructions can indicate degraded performance
  - Spilling registers to AGPRs, due to running out of VGPRs

- To determine if any Workgroup Manager “insufficient resource” stats are nonzero, you can do:
  - `rocprof-compute analyze -p workloads/problem/MI300A_A1 --block 6.2`
    - Note: This will report more than just all “insufficient resource” fields

Exercise 4: Data Access Patterns are Important to Performance

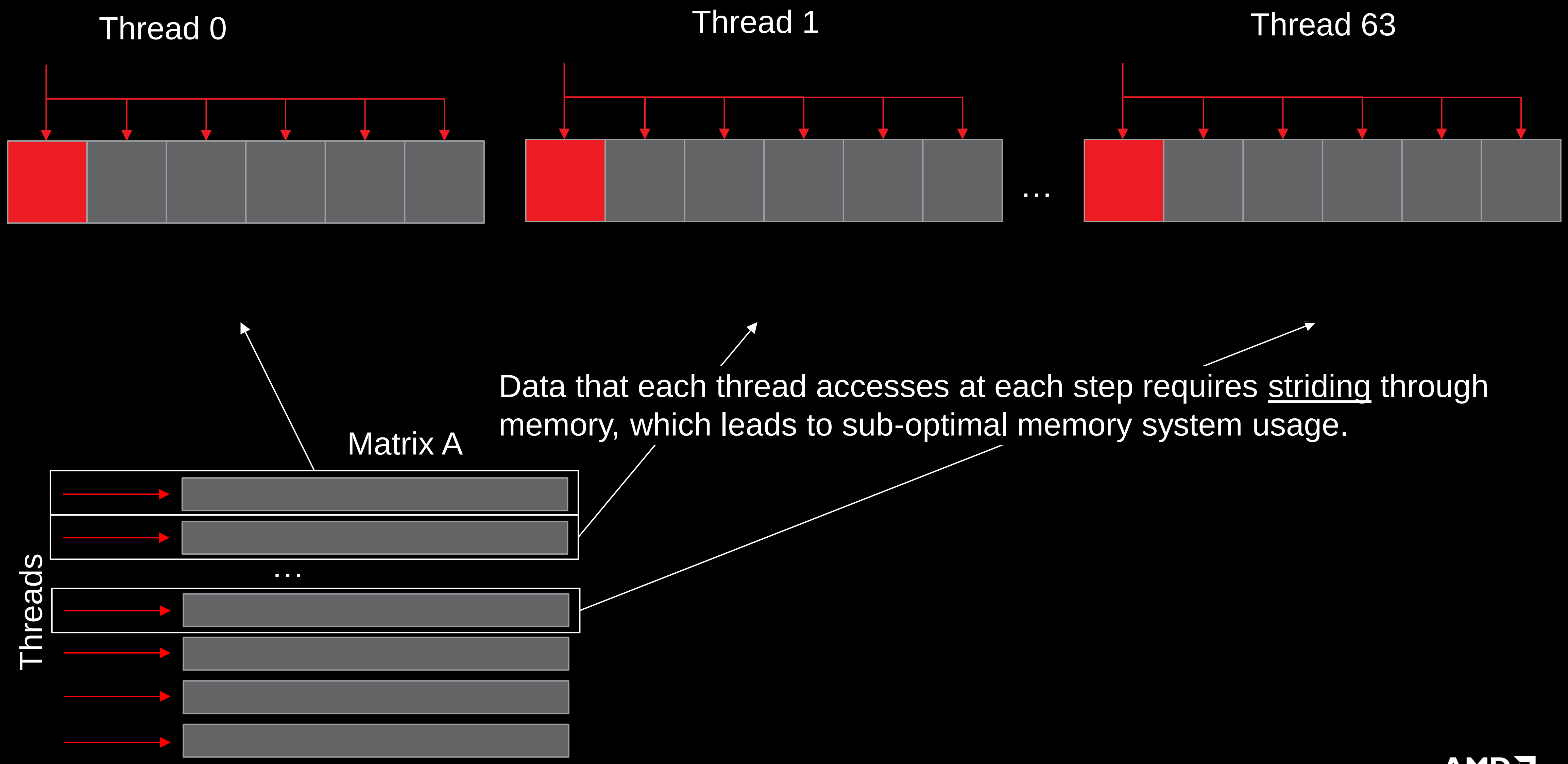
- The way in which threads access memory has a big impact on performance

- “Striding” in global memory has adverse effects on kernel performance, especially on GPUs.
  - “Strided data access patterns” lead to poor utilization of cache memory systems

- These access patterns can be difficult to spot in the code
  - They are valid methods of indexing data

- Using Rocprof-compute can quickly show if a kernel’s data access is adversarial to the caches

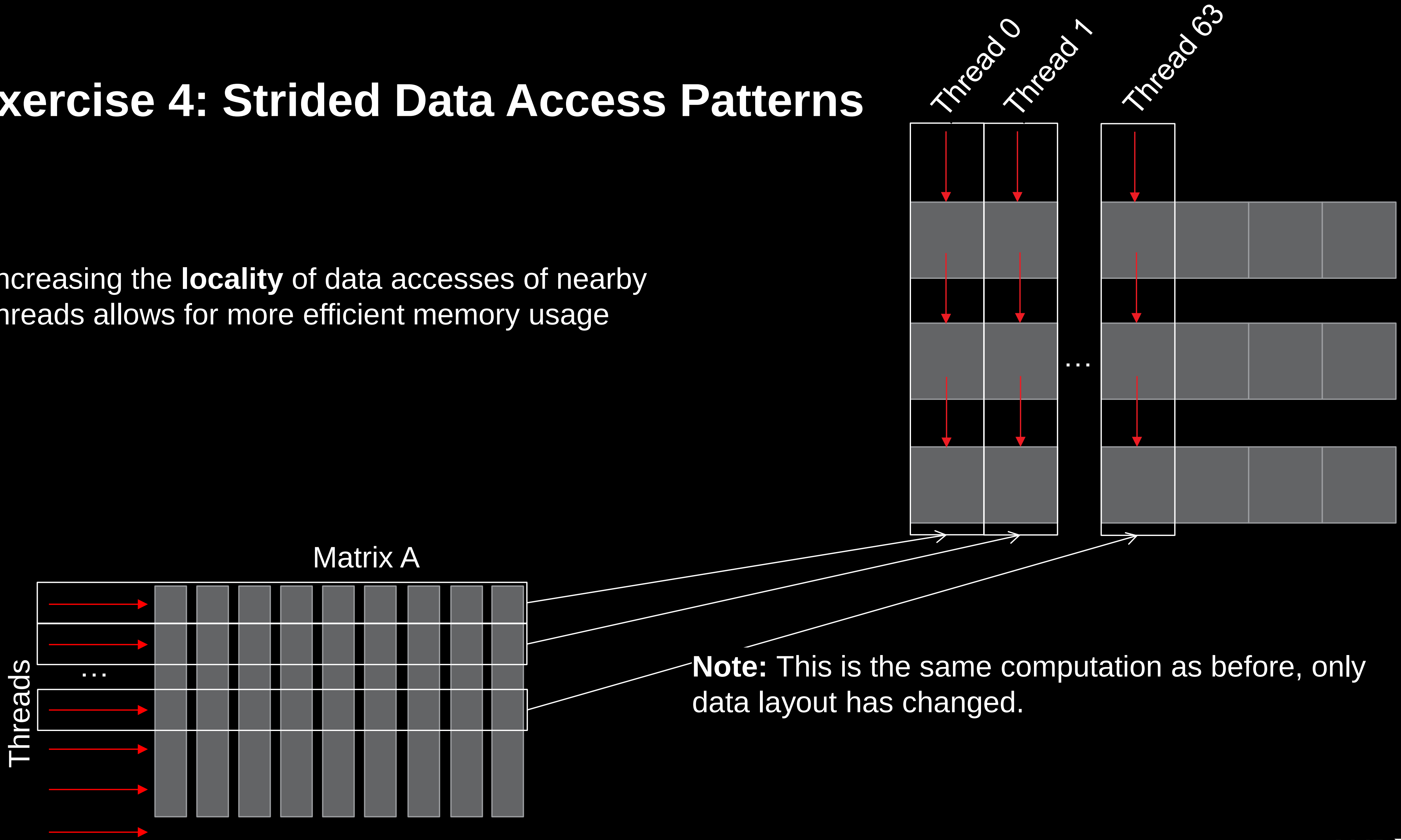
Exercise 4: What is a "Strided Data Access Pattern"?



AMD
together we advance_

Exercise 4: Strided Data Access Patterns

Increasing the **locality** of data accesses of nearby threads allows for more efficient memory usage



Note: This is the same computation as before, only data layout has changed.

Exercise 4: Diagnose a strided data access pattern

- This exercise's setup makes it very easy to change the data access pattern
  - Generally, these optimizations can have nontrivial development overhead
  - Re-conceptualizing the data's structure can be difficult

- All the solution does is re-work the indexing scheme to better use caches
  - No required change to underlying data, because all the values in y, A, and x are set to 1

- Importantly, highly contended atomics on the same global memory address is bad coding practice. This code example does that, production codes should avoid this pattern (foreshadowing)

- To get started run:
  - `rocprof-compute profile -n problem --no-roof -- ./problem.exe`
  - `rocprof-compute profile -n solution --no-roof -- ./solution.exe`

Exercise 4: Strided data access pattern – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

----

0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	100.00	100.0 (0.

~30% Slowdown?!

----

16. Vector L1 Data Cache
16.1 Speed-of-Light

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
16.1.0	Hit rate	0.01	75.0 (1061717.66%)	74.99	Pct of peak
16.1.1	Bandwidth	23.50	4.56 (-80.62%)	-18.95	Pct of peak
16.1.2	Utilization	85.08	96.69 (13.65%)	11.61	Pct of peak
16.1.3	Coalescing	25.00	25.0 (0.0%)	0.00	Pct of peak

+ ~75% in L1 hit

The solution better uses the L1, which should result in speedup. Why is the solution slower? Let's check atomic latency

----

17. L2 Cache
17.1 Speed-of-Light

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
17.1.0	Utilization	98.80	98.57 (-0.23%)	-0.23	Pct
17.1.1	Bandwidth	55.85	2.73 (-95.12%)	-53.13	Pct
17.1.2	Hit Rate	93.66	0.68 (-99.28%)	-92.99	Pct
17.1.3	L2-Fabric Read BW	912.60	698.54 (-23.46%)	-214.07	Gb/s
17.1.4	L2-Fabric Write and Atomic BW	0.01	0.01 (-0.0%)	-0.00	Gb/s

L2 Cache Hit decreases sharply

Exercise 4: Atomic latency – relevant output

```
rocprof-compute analyze -p workloads/problem/MI300A_A1 -p workloads/solution/MI300A_A1 --dispatch 1 --block 17.2.11
```

----
0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	9541187.00	12304272.0 (28.96%)	100.00	100.0 (0.

~30% Slowdown

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	4

----
17. L2 Cache
17.2 L2 - Fabric Transactions

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
17.2.11	Atomic Latency	6289.38	10098.1 (60.56%)	3808.72	6289.38	10098.1 (60.56%)	6289.38	10098.1 (60.56%)	Cycles

Solution's atomic latency is higher! This kernel is bound by atomics, not memory bandwidth

Exercise 4: Why is atomic latency higher in solution?

- In `solution.cpp`, we start hitting in the L1 cache, rather than having to go out to L2 for everything
- This reduces our memory latency, thus increasing the contention and pressure of the atomics
- This, coupled with the naïve, atomic-heavy reduction strategy, means atomics are our limiter, not cache
- This is the midpoint of the exercise, the lesson here is **not**: “use suboptimal cache access patterns”
- Let’s try to optimize our reduction strategy to use a “shuffle reduction” to reduce the atomic contention
  - You can see how this is accomplished in `mi300a_problem` and `mi300a_solution`
- Note: In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere, and reduction strategies can be costly to refactor

Exercise 4: Atomic Latency – relevant output

```
rocprof-compute analyze -p workloads/mi300a_problem/MI300A_A1 -p workloads/mi300a_solution/MI300A_A1 --dispatch 1 -block 17.2.11
```

```
INFO Analysis mode = cli
  INFO [analysis] deriving Omniperf metrics...

--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	9593149.00	12351549.0 (28.75%)	9593149.00	12351549.0 (28.75%)	9593149.00	12351549.0 (28.75%)	100.00	100.0 (0.

~30% Slowdown, still?

```
0.2 Dispatch List
```

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	4

```
--------------------------------------------------------------------------------
17. L2 Cache
17.2 L2 - Fabric Transactions
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
17.2.11	Atomic Latency	6785.81	9603.13 (41.52%)	2817.32	6785.81	9603.13 (41.52%)	6785.81	9603.13 (41.52%)	Cycles

Exact values might be slightly different, but conclusion stay the same

Solution's atomic latency is better, but still much higher!

AMD together we advance_

Exercise 4: Why is atomic latency *still* higher in solution?

- We already saw that solution uses the caches better, but this results in being bottlenecked by atomics
- We've seen that reducing atomic contention a small amount does not solve this, why?

- When atomic reduction bottleneck, and solution will always be slightly more contended than problem

- As our problem size grows, cache access and data movement should be our bottleneck

- This is the true lesson of this exercise: Profile a representative problem size!
  - Profiling problems that are too small may give you misleading optimization ideas

- Let's run `mi300a_problem` and `mi300a_solution` with larger problem sizes:
  - `rocprof-compute profile -n mi300a_problem_15 --no-roof -- ./mi300a_problem 15`
  - `rocprof-compute profile -n mi300a_solution_15 --no-roof -- ./mi300a_solution 15`

Exercise 4: Larger Problem Size – Relevant Output

```
rocprof-compute analyze -p workloads/mi300a_problem_15/MI300A_A1 -p workloads/mi300a_solution_15/MI300A_A1 --dispatch 1 --block 16.1 17.1
```

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pc
0	yax(double*, double*, double*, unsigned long long, unsigned long long, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	309917571.00	25600803.0 (-91.74%)	309917571.00	25600803.0 (-91.74%)	309917571.00	25600803.0 (-91.74%)	100.00	10

~12x speedup

```
--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
16.1.0	Hit rate	0.00	75.0 (26214512.5%)	75.00	Pct of peak
16.1.1	Bandwidth	2.89	8.76 (202.8%)	5.87	Pct of peak
16.1.2	Utilization	81.82	98.35 (20.2%)	16.53	Pct of peak
16.1.3	Coalescing	25.00	25.0 (0.0%)	0.00	Pct of peak

Similar L1 performance to the small problem size.

We finally see the speedup we expect when using a better data access pattern

```
--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit
17.1.0	Utilization	69.02	99.52 (44.19%)	30.50	Pct
17.1.1	Bandwidth	6.88	5.22 (-24.14%)	-1.66	Pct
17.1.2	Hit Rate	89.30	0.32 (-99.64%)	-88.98	Pct
17.1.3	L2-Fabric Read BW	173.83	1342.9 (672.53%)	1169.07	Gb/s
17.1.4	L2-Fabric Write and Atomic BW	0.01	0.0 (-0.0%)	-0.00	Gb/s

L2 hit rate is greatly reduced, and L2 bandwidth is greatly increased

Exercise 4: Speed-of-Light cache access statistics

- The command below will show high-level details about L1 and L2 cache accesses:
  - `rocprof-compute analyze -p workloads/problem/MI300A_A1 --dispatch 1 --block 16.1 17.1`
- Ensuring better data locality will generally provide better performance
- In this case, we start hitting in the L1 cache, rather than having to go out to L2 for everything
- If you increase your cache efficiency but are running a small problem, you can check atomic latency:
  - `rocprof-compute analyze –p workloads/problem/MI300A_A1 --dispatch 1 --block 17.2.11`
- Note: In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere

Rocprofiler-compute tips

- Filtering by kernel name and metrics during rocprof-compute profile will cut down on profiling time
  - rocprof-compute profile -k “<kernel1>” “<kernel2>” filters two kernel names
  - Surrounding kernel name in quotes allows spaces to appear in your kernel search string
  - Rocprofiler-compute applies wildcard automatically, so only unique kernel names substring required
- Use a subset of metrics for rocprof-compute profile to reduce the number of rocprof runs
  - rocprof-compute profile --block SQ SQC –n <workload name> -- ./benchmark.sh
  - rocprof-compute profile --help displays all block strings you can filter by
  - Performance model doc goes over some of the meaning behind lower-level hardware units and metrics
- MPI/srun support still brittle, safest way is to use node interactively and run only with 1 MPI rank
- Don’t know where to start? → Easy things to check:
  - Are all the CUs being used? → If not, more parallelism is required (for most of the cases)
  - Are all the VGPRs being spilled? → Try smaller workgroup sizes
  - Is the code Integer limited? → Try reducing the integer ops, usually in the index calculation

Summary

- Rocprof-compute is a tool that collects many counters automatically
- It can create roofline analysis to understand how efficient are your kernels
- It displays a lot of metrics regarding your kernels, however, it is required to know more about your kernel
- It does not have learning curve to start running it, but requies knowledge for the analysis
- It supports Grafana, standalone GUI, and CLI
- Includes several features such as:
  - System Speed-of-Light Panel
  - Memory Chart Analysis Panel
  - Vector L1D Cache Panel
  - Shader Processing Input (SPI) Panel

Questions?

DISCLAIMERS AND ATTRIBUTIONS

AMD