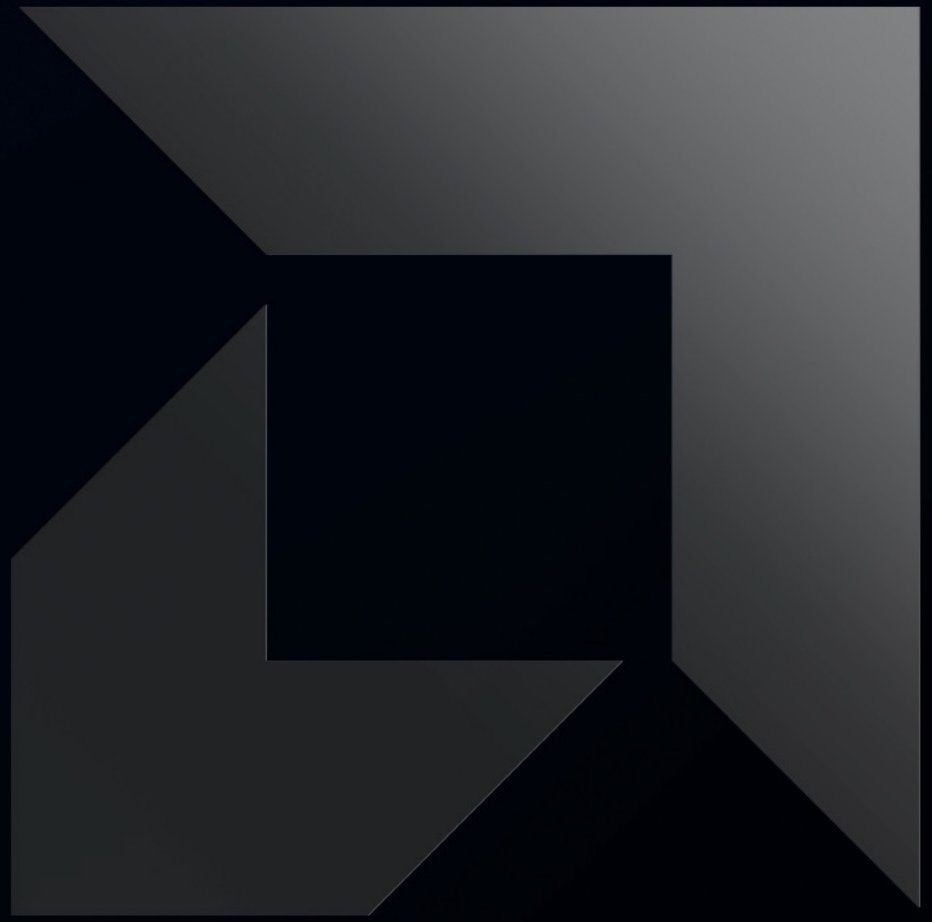


Introduction to Rocprofiler-compute

Presenter: Sam Antao

LUMI Advanced Training

Nov 12th, 2026

AMD

together we advance_

[Public]

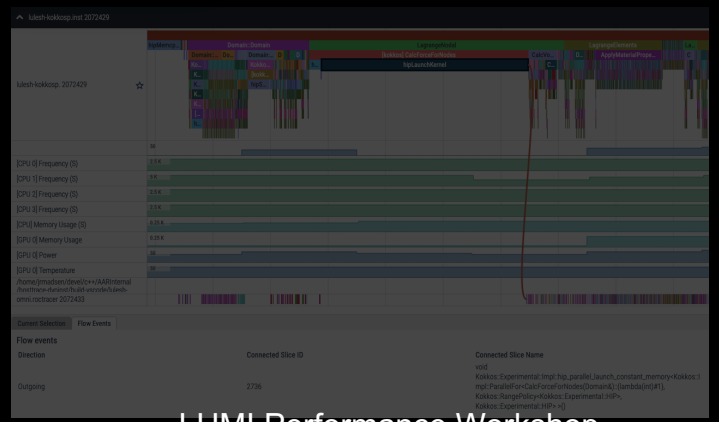
Background – AMD Profilers

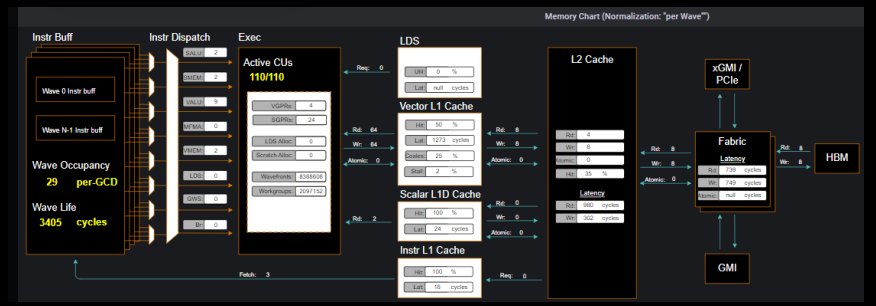
ROC-profiler (rocprof)			
Hardware Counters	Raw collection of GPU counters and traces		
	Counter collection with user input files	Counter results printed to a CSV	
Traces and timelines	Trace collection support for		
	CPU copy	HIP API	HSA API	GPU Kernels
Visualisation	Traces visualized with Perfetto		

Rocprofiler-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	Visualization		
Supports	Speed of Light	Memory chart	Rooflines	Kernel comparison
Visualisation	With Grafana or standalone GUI			

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





AMD
together we advance_

[Public]

Rocprofiler-compute is distributed with ROCm

Development now part of ROCm Systems repository:
https://github.com/ROCm/rocm-systems/tree/develop/projects/rocprofiler-compute

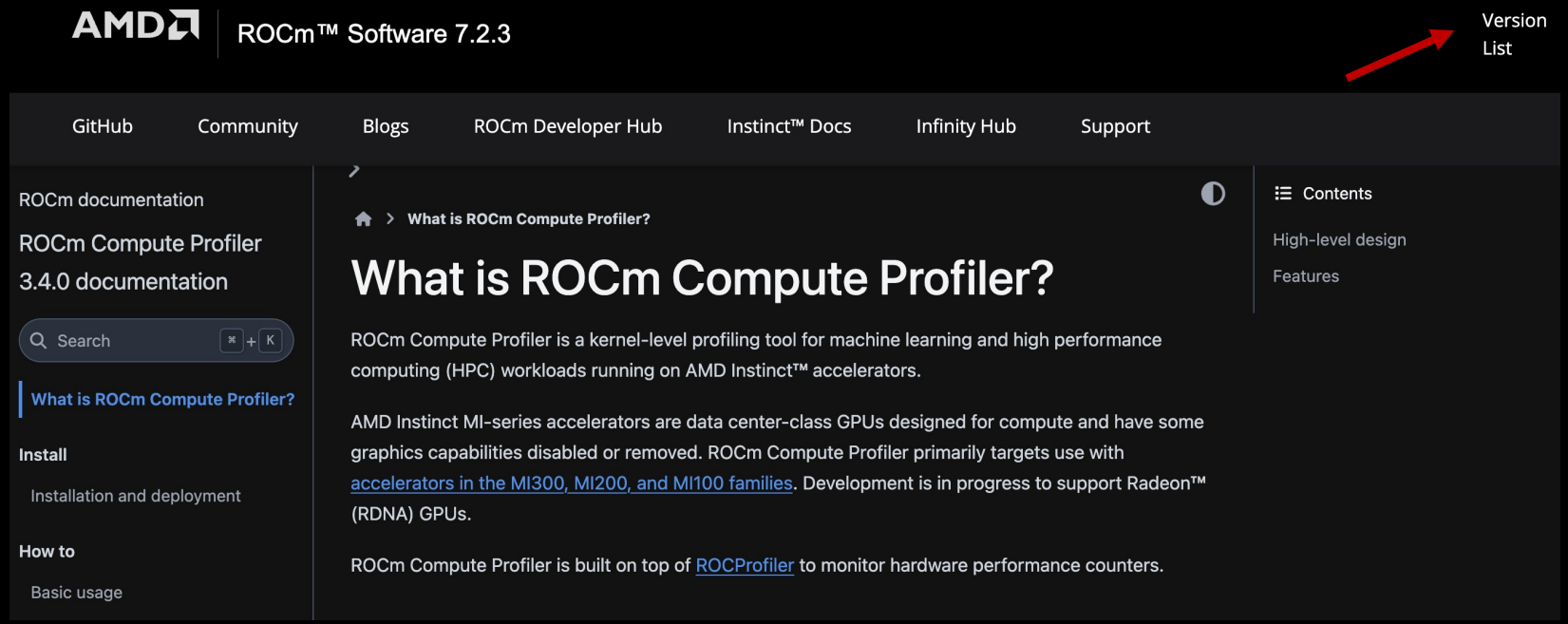
Full documentation part of official ROCm documentation
https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/what-is-rocprof-compute.html

- Package rocprofiler-compute not part of ROCm metapackages
- Not installed with default ROCm on LUMI

```
module load
    LUMI/25.03
    partition/G
    rocm/6.3.4-extras
```

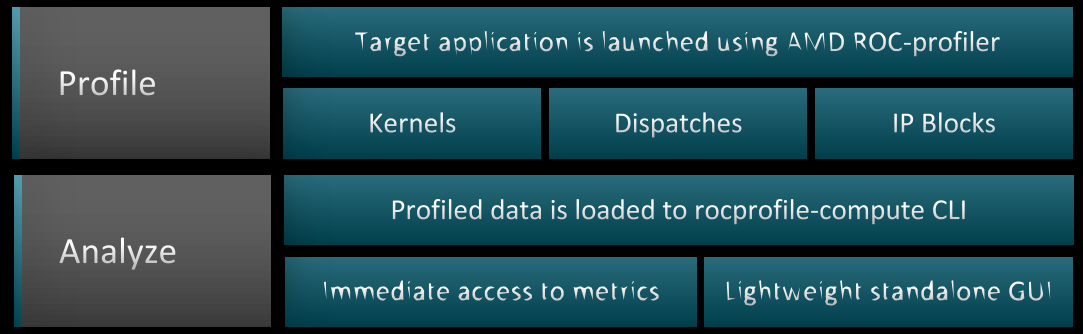
Python Dependencies

- Python™ (>=3.8)
  - pip install --force-reinstall -r $ROCM_PATH/libexec/rocprofiler-compute/requirements.txt



[Public]

Rocprofiler-compute modes



Basic command-line syntax:

Profile:

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

```
$ ROCPROF=rocprofv3 \
  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/MI210/> --gui
```

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

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

For problems, create an issue here: https://github.com/ROCm/rocm-systems/issues
Documentation: https://rocm.docs.amd.com/projects/rocprofiler-compute

[Public]

`Rocprof-compute profile` Args. to Reduce Profiling Overhead

- **Runtime Filtering**
  `--kernel`, `--blocks`, `--dispatch`

- The `-k` <kernel> flag allows for kernel filtering, which is compatible with the current rocprof utility.
- The `-d` <dispatch> flag allows for dispatch ID filtering, which is compatible with the current rocprof utility.
- The `-b` <ipblocks> allows system profiling on one or more selected IP blocks to speed up the profiling process. One can gradually incorporate more IP blocks, without overwriting performance data acquired on other IP blocks.

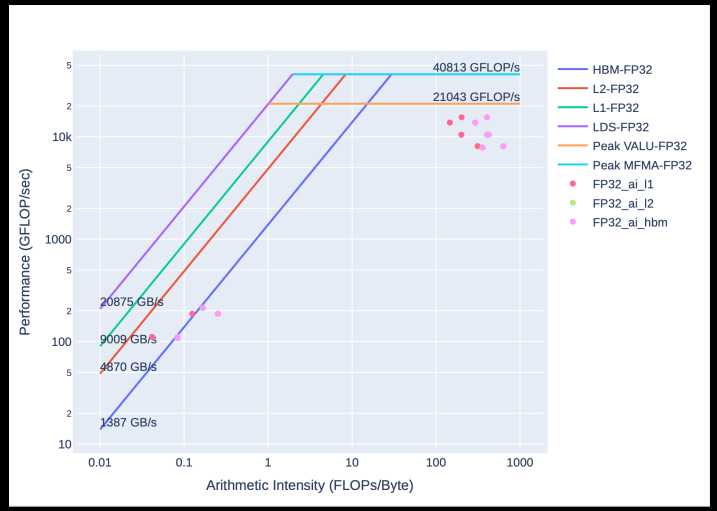
Note: the `-k/--kernel` flag takes a search string in `rocprof-compute profile`

[Public]

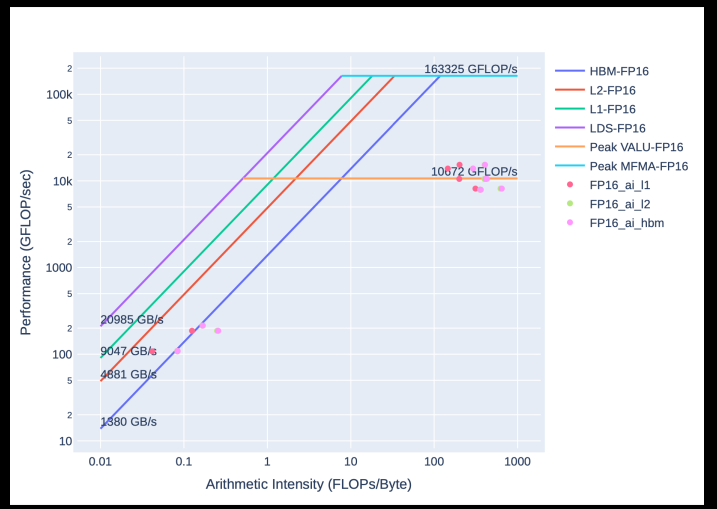
Rocprof-compute profile Arguments to Generate Roofline PDFs

- Runtime Filtering
  `--kernel`, `--blocks`, `--dispatch`
- **Standalone Roofline Analysis**
  `--roof-only, --kernel-names`

FP32



FP16



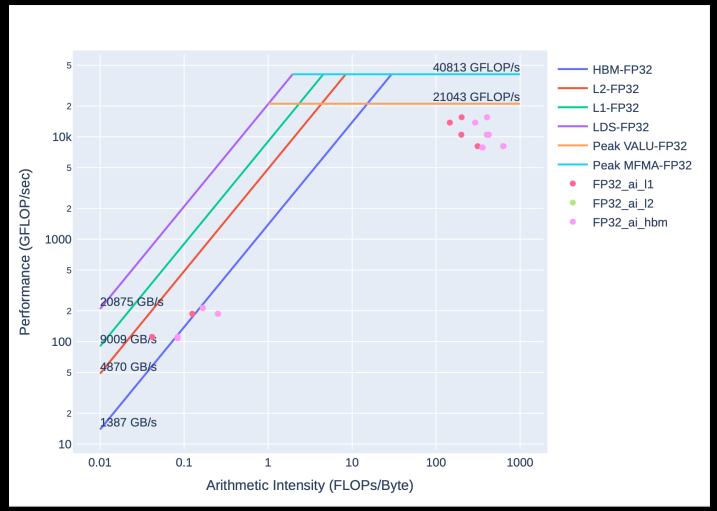
The above plots are saved as PDF output when the --roof-only option is used. Other FLOP types can be selected

[Public]

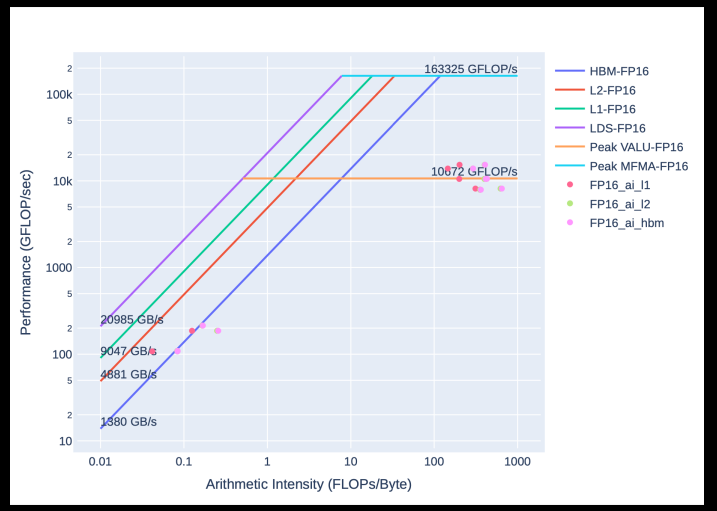
`Rocprof-compute profile` Arg. to Turn Off Roofline Benchmarking

- Runtime Filtering
  `--kernel`, `--block`, `--dispatch`
- Standalone Roofline Analysis
  `--roof-only, --kernel-names`
- **No roofline analysis**
  `--no-roof`

FP32



FP16



--no-roof will skip the roofline microbenchmark and omit roofline from output

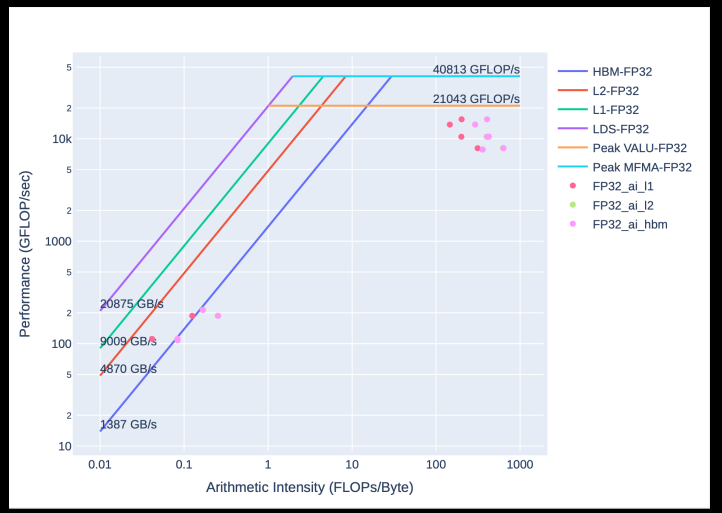
[Public]

Rocprof-compute profile Arg. to define FLOP type

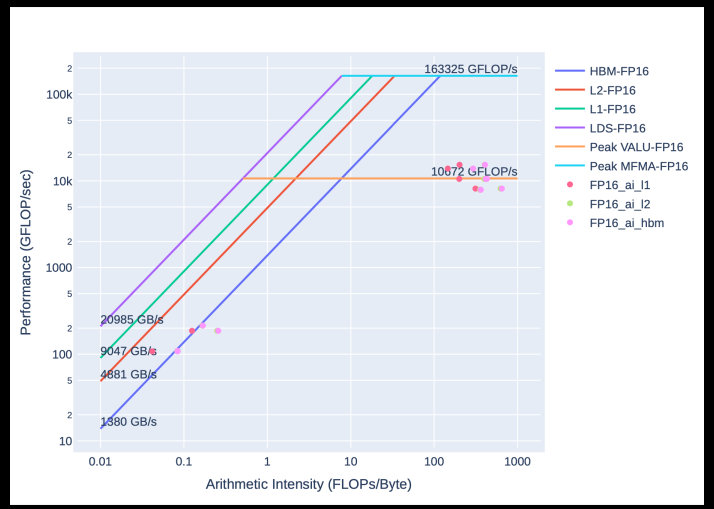
- Runtime Filtering
  `--kernel`, `--block`, `--dispatch`
- Standalone Roofline Analysis
  `--roof-only, --kernel-names`
- No roofline analysis
  `--no-roof`
- **Control roofline relevant types**
  `--roofline-data-type`

FP64, FP32, FP16

FP32



FP16



FP64 down to INT8 can be selected – some native types not available on MI250X

[Public]

Rocprof-compute profiling

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

```
$ wget https://github.com/ROCm/rocm-systems/raw/develop/projects/rocprofiler-compute/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 -i 10
…
   INFO Rocprofiler-Compute version: 3.1.1
   INFO Profiler choice: rocprofv3
   INFO Path: /…/workloads/vcopy_all/MI210
   INFO Target: MI210
   INFO Command: ./vcopy -n 1048576 -b 256 -i 10
   INFO Kernel Selection: None
   INFO Dispatch Selection: None
   INFO Hardware 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.

[Public]

Rocprof-compute analyze Arguments to Start With

- **List top kernels or view list of metrics**
  `--list-stats`, `--list-metrics`

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --list-stats
```

```
INFO:root:Analysis mode = cli
   INFO Analysis mode = cli
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
```

Detected Kernels (sorted descending by duration)

	Kernel_Name
0	k03_fp64_mandelbrot(double const*, double const*, double*, int) [clone .kd]
1	k01_fp64_poly(double const*, double*, int) [clone .kd]
2	k04_fp32_transcendental(float const*, float*, int) [clone .kd]
3	k06_fp32_newton(float const*, float*, int) [clone .kd]
4	k10_fp16_sigmoid(__half const*, __half*, int) [clone .kd]
5	k08_fp16_fma(__half const*, __half*, int) [clone .kd]
6	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]
7	k07_fp32_saxpy(float const*, float*, float, int) [clone .kd]
8	k05_fp32_vecadd(float const*, float const*, float*, int) [clone .kd]
9	k09_fp16_scale(__half const*, __half*, __half, int) [clone .kd]
10	init_half(__half*, float, int) [clone .kd]

Dispatch list

	Dispatch_ID	Kernel_Name	GPU_ID
0	0	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
1	1	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
2	2	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
3	3	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
4	4	k01_fp64_poly(double const*, double*, int) [clone .kd]	4

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --list-metrics gfx90a
```

```
INFO:root:Analysis mode = cli
   INFO Analysis mode = cli
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
0 -> Top Stats
1 -> System Info
2 -> System Speed-of-Light
        2.1 -> Speed-of-Light
                2.1.0 -> VALU FLOPs
                2.1.1 -> VALU IOPs
                2.1.2 -> MFMA FLOPs (F8)
                2.1.3 -> MFMA FLOPs (BF16)
                2.1.4 -> MFMA FLOPs (F16)
                2.1.5 -> MFMA FLOPs (F32)
                2.1.6 -> MFMA FLOPs (F64)
                2.1.7 -> MFMA IOPs (Int8)
                2.1.8 -> Active CUs
                2.1.9 -> SALU Utilization
                2.1.10 -> VALU Utilization
                2.1.11 -> MFMA Utilization
                2.1.12 -> VMEM Utilization
                2.1.13 -> Branch Utilization
                2.1.14 -> VALU Active Threads
                2.1.15 -> IPC
                2.1.16 -> Wavefront Occupancy
                2.1.17 -> Theoretical LDS Bandwidth
                2.1.18 -> LDS Bank Conflicts/Access
                2.1.19 -> vL1D Cache Hit Rate
                2.1.20 -> vL1D Cache BW
                2.1.21 -> L2 Cache Hit Rate
                2.1.22 -> L2 Cache BW
                2.1.23 -> L2-Fabric Read BW
                2.1.24 -> L2-Fabric Write BW
                2.1.25 -> L2-Fabric Read Latency
                2.1.26 -> L2-Fabric Write Latency
                2.1.27 -> sL1D Cache Hit Rate
                2.1.28 -> sL1D Cache BW
                2.1.29 -> L1I Hit Rate
                2.1.30 -> L1I BW
                2.1.31 -> L1I Fetch Latency
3 -> Memory Chart
        3.1 -> Memory Chart
                3.1.0 -> Wavefront Occupancy
                3.1.1 -> Wave Life
                3.1.2 -> SALU
                3.1.3 -> SMEM
                3.1.4 -> VALU
```

Output from the --list-stats and --list-metric options, showing top kernels and available metrics

[Public]

Rocprof-compute analyze Arguments to Filter Kernels and GPUs

- List top kernels or view list of metrics
  `--list-stats`, `--list-metrics`
- **Filter available kernels, dispatches, gpu-ids**
  `--kernel`, `--dispatch`, `--gpu-id`

Note: the `-k/--kernel` flag takes an index given by `--list-stats` in `Rocprof-compute analyze`, and aggregates stats by kernel name

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --kernel 0
```

```
INFO:root:Analysis mode = cli
   INFO Analysis mode = cli
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
```

0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct	S
0	k03_fp64_mandelbrot(double const*, double const*, double*, int) [clone .kd]	25.00	427069652.00	17082786.08	17088188.00	24.08	*
1	k01_fp64_poly(double const*, double*, int) [clone .kd]	25.00	349167579.00	13966703.16	13987062.00	19.69	
2	k04_fp32_transcendental(float const*, float*, int) [clone .kd]	25.00	304456329.00	12178253.16	12182740.00	17.17	
3	k06_fp32_newton(float const*, float*, int) [clone .kd]	25.00	270105239.00	10804209.56	10805457.00	15.23	
4	k10_fp16_sigmoid(__half const*, __half*, int) [clone .kd]	25.00	259170507.00	10366820.28	10366897.00	14.61	
5	k08_fp16_fma(__half const*, __half*, int) [clone .kd]	25.00	128172047.00	5126881.88	5129768.00	7.23	
6	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	25.00	14836982.00	593479.28	592640.00	0.84	
7	k07_fp32_saxpy(float const*, float*, float, int) [clone .kd]	25.00	7800011.00	312000.44	311840.00	0.44	
8	k05_fp32_vecadd(float const*, float const*, float*, int) [clone .kd]	25.00	7601612.00	304064.48	304641.00	0.43	
9	k09_fp16_scale(__half const*, __half*, __half, int) [clone .kd]	25.00	4490247.00	179609.88	179361.00	0.25	

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	0	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
1	1	k01_fp64_poly(double const*, double*, int) [clone .kd]	4

Filtered output from the --kernel option isolating kernel at index 0

[Public]

`Rocprof-compute analyze` Arg. to Only Show Specific Statistics

- List top kernels or view list of metrics
  `--list-stats`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`
- **Filter by metric id(s)**
  `--block`

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --dispatch 2 --block 5
```

```
INFO:root:Analysis mode = cli
   INFO Analysis mode = cli
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
```

0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	k01_fp64_poly(double const*, double*, int) [clone .kd]	1.00	13914742.00	13914742.00	13914742.00	100.00

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	2	k01_fp64_poly(double const*, double*, int) [clone .kd]	4

5. Command Processor (CPC/CPF)
5.1 Command Processor Fetcher

Metric_ID	Metric	Avg	Min	Max	Unit
5.1.0	CPF Utilization	100.00	100.00	100.00	Pct
5.1.1	CPF Stall	0.01	0.01	0.01	Pct
5.1.2	CPF-L2 Utilization	0.03	0.03	0.03	Pct
5.1.3	CPF-L2 Stall	0.00	0.00	0.00	Pct
5.1.4	CPF-UTCL1 Stall	0.01	0.01	0.01	Pct

5.2 Packet Processor

Metric_ID	Metric	Avg	Min	Max	Unit
5.2.0	CPC Utilization	100.00	100.00	100.00	Pct
5.2.1	CPC Stall Rate	0.21	0.21	0.21	Pct
5.2.2	CPC Packet Decoding Utilization	0.74	0.74	0.74	Pct

Filtering output to isolate data table at index 5

[Public]

Rocprof-compute analyze Arguments Units and Normalization

- List top kernels or view list of metrics
  `--list-kernels`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`
- Filter by block/metric id(s)
  `--block`
- **Change normalization unit, time unit, or decimal**
  `--normal-unit`, `--time-unit`, `--decimal`

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --dispatch 2 --block 7.2 --decimal 1 --time-unit ms
```

```
INFO:root:Analysis mode = cli
   INFO Analysis mode = cli
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
```

0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Sum(ms)	Mean(ms)	Median(ms)	Pct
0	k01_fp64_poly(double const*, double*, int) [clone .kd]	1.0	13.9	13.9	13.9	100.0

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	2	k01_fp64_poly(double const*, double*, int) [clone .kd]	4

7. Wavefront
7.2 Wavefront Runtime Stats

Metric_ID	Metric	Avg	Min	Max	Unit
7.2.0	Kernel Time (Nanosec)	13914742.0	13914742.0	13914742.0	Ns
7.2.1	Kernel Time (Cycles)	16376877.0	16376877.0	16376877.0	Cycle
7.2.2	Instructions per wavefront	3673.0	3673.0	3673.0	Instr/wavefront
7.2.3	Wave Cycles	109159.7	109159.7	109159.7	Cycles per wave
7.2.4	Dependency Wait Cycles	1698.0	1698.0	1698.0	Cycles per wave
7.2.5	Issue Wait Cycles	92878.3	92878.3	92878.3	Cycles per wave
7.2.6	Active Cycles	14484.0	14484.0	14484.0	Cycles per wave
7.2.7	Wavefront Occupancy	3495.5	3495.5	3495.5	Wavefronts

Output showing the default normalization and time ms, with single decimal digit.

[Public]

Rocprof-compute analyze Arguments to Compare Workloads

- **Baseline Analysis**

  `--path <workload1_path> --path <workload2_path>`

Note: rocprof-compute averages metrics over several dispatches - sometime is useful to target a single dispatch to make sure the comparison is sensible. A change can have different effects in different instances of a kernel.

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
2.1.0	VALU FLOPs	113.02	95.0 (-15.94%)	-18.02	Gflop	23936.00	23936.0 (0.0%)	0.47	0.4 (-16.94%)
2.1.1	VALU IOPs	508.61	427.5 (-15.95%)	-81.10	Giop	23936.00	23936.0 (0.0%)	2.12	1.79 (-16.0%)
2.1.2	MFMA FLOPs (F8)		0.0 (0.0%)	0.00	Gflop		0.0 (0.0%)		0.0 (0.0%)
2.1.3	MFMA FLOPs (BF16)	0.00	0.0 (0.0%)	0.00	Gflop	191488.00	191488.0 (0.0%)	0.00	0.0 (0.0%)
2.1.4	MFMA FLOPs (F16)	0.00	0.0 (0.0%)	0.00	Gflop	191488.00	191488.0 (0.0%)	0.00	0.0 (0.0%)
2.1.5	MFMA FLOPs (F32)	0.00	0.0 (0.0%)	0.00	Gflop	47872.00	47872.0 (0.0%)	0.00	0.0 (0.0%)
2.1.6	MFMA FLOPs (F64)	0.00	0.0 (0.0%)	0.00	Gflop	47872.00	47872.0 (0.0%)	0.00	0.0 (0.0%)
2.1.7	MFMA IOPs (Int8)	0.00	0.0 (0.0%)	0.00	Giop	191488.00	191488.0 (0.0%)	0.00	0.0 (0.0%)
2.1.8	Active CUs	110.00	98.0 (-10.91%)	-12.00	Cus	110.00	110.0 (0.0%)	100.00	89.09 (-10.91%)
2.1.9	SALU Utilization	3.06	2.57 (-16.01%)	-0.49	Pct	100.00	100.0 (0.0%)	3.06	2.57 (-16.01%)
2.1.10	VALU Utilization	7.14	5.99 (-16.1%)	-1.15	Pct	100.00	100.0 (0.0%)	7.14	5.99 (-16.1%)
2.1.11	MFMA Utilization	0.00	0.0 (0.0%)	0.00	Pct	100.00	100.0 (0.0%)	0.00	0.0 (0.0%)
2.1.12	VMEM Utilization	1.53	1.28 (-16.34%)	-0.25	Pct	100.00	100.0 (0.0%)	1.53	1.28 (-16.34%)
2.1.13	Branch Utilization	1.02	0.86 (-15.68%)	-0.16	Pct	100.00	100.0 (0.0%)	1.02	0.86 (-15.68%)
2.1.14	VALU Active Threads	64.00	64.0 (0.0%)	0.00	Threads	64.00	64.0 (0.0%)	100.00	100.0 (0.0%)
2.1.15	IPC	0.15	0.15 (0.0%)	0.00	Instr/cycle	5.00	5.0 (0.0%)	2.91	2.92 (0.34%)
2.1.16	Wavefront Occupancy	3105.77	981.4 (-68.4%)	-2124.37	Wavefronts	3520.00	3520.0 (0.0%)	88.23	27.88 (-68.4%)
2.1.17	Theoretical LDS Bandwidth	0.00	0.0 (0.0%)	0.00	Gb/s	23936.00	23936.0 (0.0%)	0.00	0.0 (0.0%)
2.1.18	LDS Bank Conflicts/Access		0.0 (0.0%)	0.00	Conflicts/access	32.00	32.0 (0.0%)		0.0 (0.0%)
2.1.19	vL1D Cache Hit Rate	50.00	50.0 (0.0%)	0.00	Pct	100.00	100.0 (0.0%)	50.00	50.0 (0.0%)
2.1.20	vL1D Cache BW	2712.56	2280.02 (-15.95%)	-432.54	Gb/s	11968.00	11968.0 (0.0%)	22.67	19.05 (-15.93%)
2.1.21	L2 Cache Hit Rate	31.57	27.59 (-12.61%)	-3.98	Pct	100.00	100.0 (0.0%)	31.57	27.59 (-12.61%)
2.1.22	L2 Cache BW	1982.02	1574.42 (-20.56%)	-407.60	Gb/s	3481.60	3481.6 (0.0%)	56.93	45.22 (-20.57%)
2.1.23	L2-Fabric Read BW	904.20	760.02 (-15.95%)	-144.19	Gb/s	1638.40	1638.4 (0.0%)	55.19	46.39 (-15.95%)
2.1.24	L2-Fabric Write BW	448.81	377.68 (-15.85%)	-71.13	Gb/s	1638.40	1638.4 (0.0%)	27.39	23.05 (-15.84%)

[Public]

Rocprof-compute analyze Argument to Launch Standalone GUI

- Baseline Analysis
  `--path <workload1_path> --path <workload2_path>`
- **Launch a standalone HTML page from terminal**
  `--gui <port>`

```
$ rocprof-compute analyze -p workloads/run_25/MI210/ --gui 64321

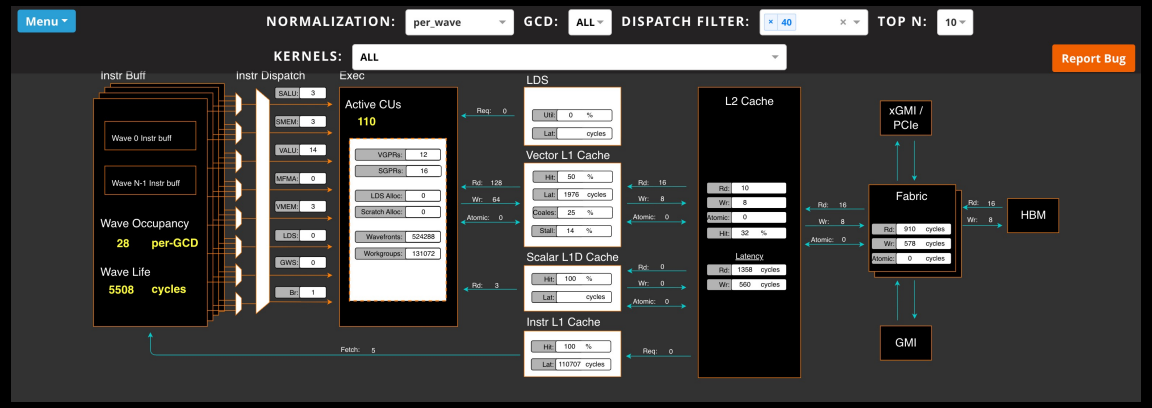
INFO:root:Analysis mode = web_ui
   INFO Analysis mode = web_ui
INFO:root:[analysis] deriving rocprofiler-compute metrics...
   INFO [analysis] deriving rocprofiler-compute metrics...
Dash is running on http://0.0.0.0:64321/

INFO:dash.dash:Dash is running on http://0.0.0.0:64321/

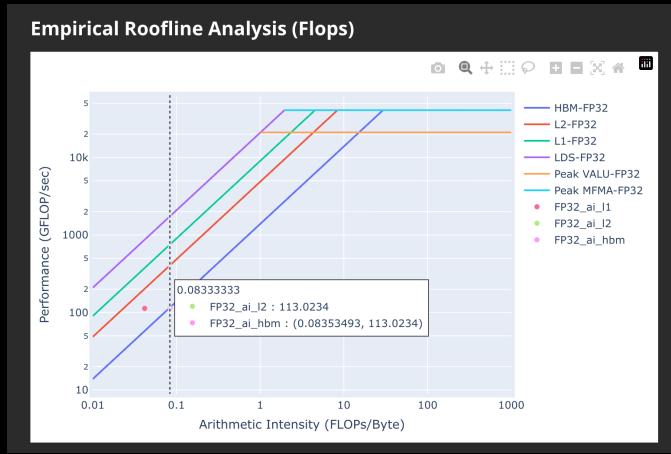
   INFO Dash is running on http://0.0.0.0:64321/

 * Serving Flask app 'rocprof_compute_analyze.analysis_webui'
 * Debug mode: off
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:64321
 * Running on http://10.253.0.149:64321
   INFO WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:64321
 * Running on http://10.253.0.149:64321
INFO:werkzeug:Press CTRL+C to quit
   INFO Press CTRL+C to quit
```

Terminal output from the --gui option with full port forwarding info



The above webpage is launched when the --gui option is used



Roofline presented for the data type selected – FP32 in this example.



Key Insights from Rocprof-compute Analyzer

[Public]

High level Metrics

- System Info

1. System Info

index	Info
workload_name	run_25
command	./gpu_kernels 25
ip_blocks	SQ|LDS|SQC|TA|TD|TCP|TCC|SPI|CPC|CPF|roofline
timestamp	Sat 09 May 2026 03:13:31 PM (EEST)
version	3
hostname	nid005002
cpu_model	AMD EPYC 7A53 64-Core Processor
sbios	HPE2.2.0
linux_distro	SUSE Linux Enterprise Server 15 SP6
linux_kernel_version	6.4.0-150600.23.73_15.0.14-cray_shasta_c
amd_gpu_kernel_version	
cpu_memory	527579908
gpu_memory	
rocm_version	6.4.4-129
vbios	113-D65201-046
compute_partition	
memory_partition	
gpu_series	MI200
gpu_model	MI210
gpu_arch	gfx90a
gpu_l1	16
gpu_l2	8192
cu_per_gpu	110
simd_per_cu	4

Detailed system info for each app is collected by default

[Public]

High level Metrics

- System Info
- System Speed-of-Light

2. System Speed-of-Light

2.1 Speed-of-Light

Metric	Avg	Unit	Peak	Pct of Peak
VALU FLOPs	15553.73	Gflop	23936.00	64.98
VALU IOPs	16.88	Giop	23936.00	0.07
MFMA FLOPs (F8)		Gflop		
MFMA FLOPs (BF16)	0.00	Gflop	191488.00	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	110.00	Cus	110.00	100.00
SALU Utilization	4.05	Pct	100.00	4.05
VALU Utilization	99.77	Pct	100.00	99.77
MFMA Utilization	0.00	Pct	100.00	0.00
VMEM Utilization	0.06	Pct	100.00	0.06
Branch Utilization	1.51	Pct	100.00	1.51
VALU Active Threads	64.00	Threads	64.00	100.00
IPC	1.07	Instr/cycle	5.00	21.41
Wavefront Occupancy	3495.53	Wavefronts	3520.00	99.30
Theoretical LDS Bandwidth	0.00	Gb/s	23936.00	0.00
LDS Bank Conflicts/Access		Conflicts/access	32.00	
vL1D Cache Hit Rate	50.00	Pct	100.00	50.00
vL1D Cache BW	77.17	Gb/s	11968.00	0.64
L2 Cache Hit Rate	33.54	Pct	100.00	33.54
L2 Cache BW	58.06	Gb/s	3481.60	1.67
L2-Fabric Read BW	19.29	Gb/s	1638.40	1.18
L2-Fabric Write BW	19.11	Gb/s	1638.40	1.17
L2-Fabric Read Latency	228.48	Cycles		
L2-Fabric Write Latency	137.42	Cycles		
sL1D Cache Hit Rate	99.99	Pct	100.00	99.99
sL1D Cache BW	7.23	Gb/s	6092.80	0.12
L1I Hit Rate	100.00	Pct	100.00	100.00
L1I BW	1726.58	Gb/s	6092.80	28.34
L1I Fetch Latency	16.02	Cycles		

Calls attention to high level performance stats to preview overall application performance

[Public]

High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats

0. Top Stats

0.1 Top Kernels

Kernel_Name	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
k03_fp64_mandelbrot(double const*, double const*, double*, int) [clone .kd]	25.00	427069652.00	17082786.08	17088188.00	24.08
k01_fp64_poly(double const*, double*, int) [clone .kd]	25.00	349167579.00	13966703.16	13987062.00	19.69
k04_fp32_transcendental(float const*, float*, int) [clone .kd]	25.00	304456329.00	12178253.16	12182740.00	17.17
k06_fp32_newton(float const*, float*, int) [clone .kd]	25.00	270105239.00	10804209.56	10805457.00	15.23
k10_fp16_sigmoid(__half const*, __half*, int) [clone .kd]	25.00	259170507.00	10366820.28	10366897.00	14.61
k08_fp16_fma(__half const*, __half*, int) [clone .kd]	25.00	128172047.00	5126881.88	5129768.00	7.23
k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	25.00	14836982.00	593479.28	592640.00	0.84
k07_fp32_saxpy(float const*, float*, float, int) [clone .kd]	25.00	7800011.00	312000.44	311840.00	0.44
k05_fp32_vecadd(float const*, float const*, float*, int) [clone .kd]	25.00	7601612.00	304064.48	304641.00	0.43
k09_fp16_scale(__half const*, __half*, __half, int) [clone .kd]	25.00	4490247.00	179609.88	179361.00	0.25
init_half(__half*, float, int) [clone .kd]	3.00	532001.00	177333.67	176961.00	0.03

0.2 Dispatch List

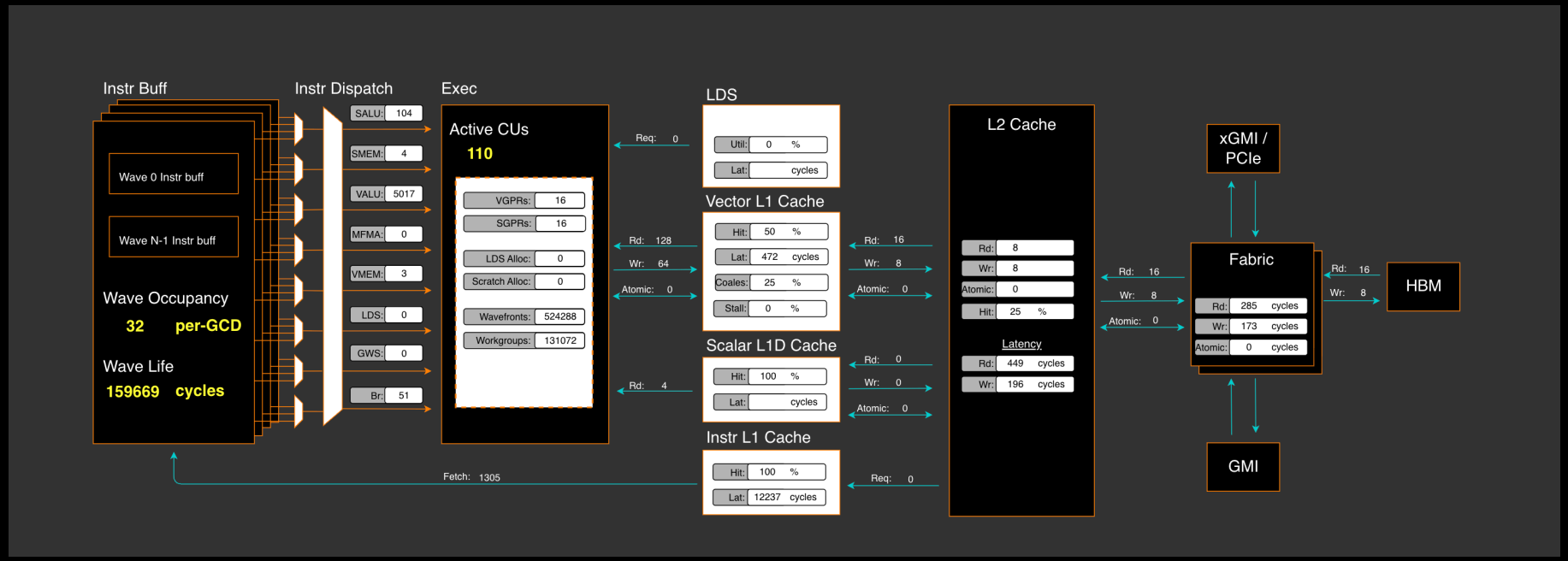
Dispatch_ID	Kernel_Name	GPU_ID
23	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
24	k01_fp64_poly(double const*, double*, int) [clone .kd]	4
25	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
26	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
27	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
28	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
29	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
30	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
31	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
32	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
33	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4
34	k02_fp64_triad(double const*, double const*, double*, double, int) [clone .kd]	4

Preview performance of top N kernels and individual kernel invocations (dispatches)

[Public]

High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis

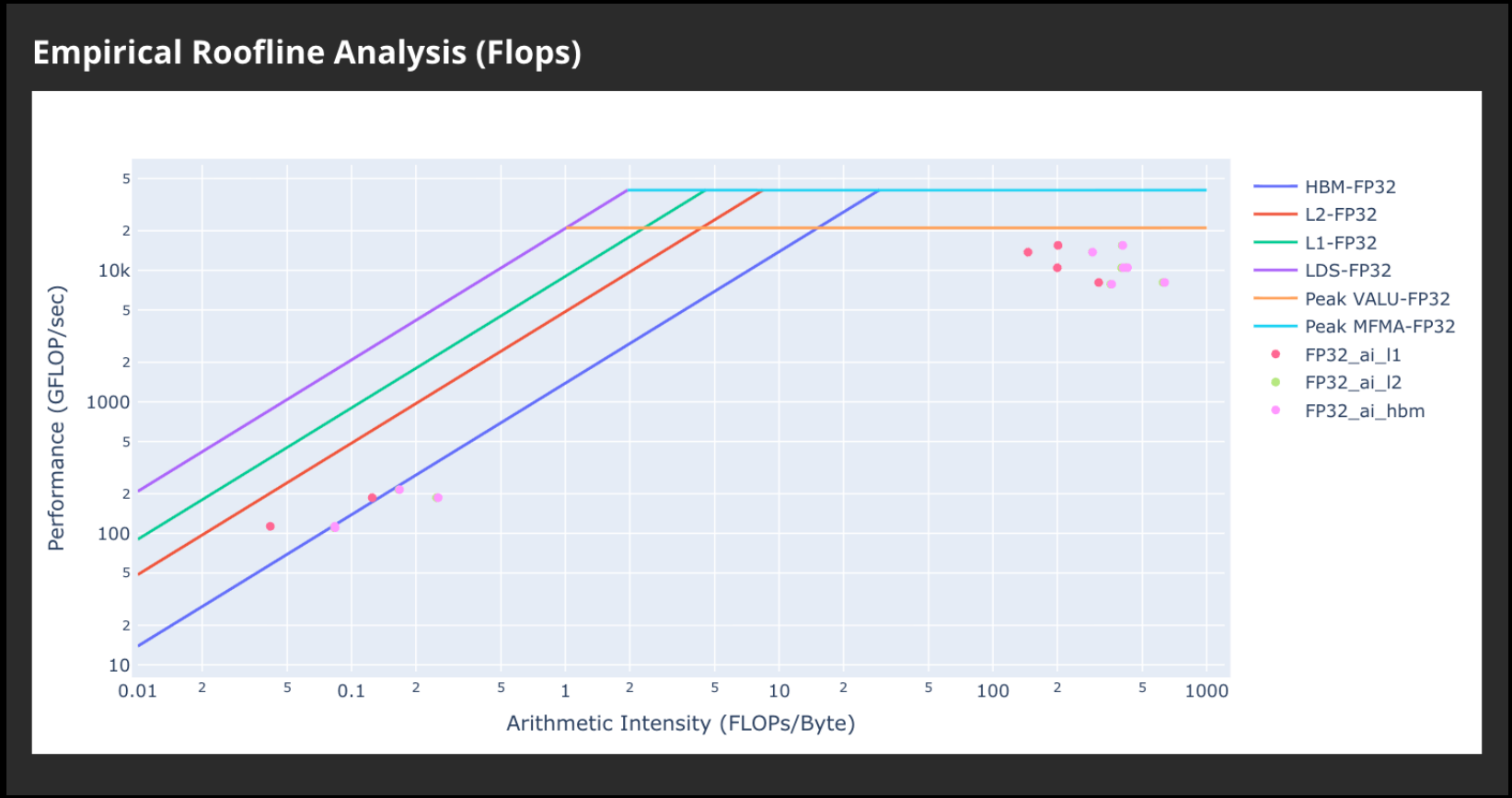


Illustrate data movement and performance on key components of target architecture

[Public]

High level Metrics

- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis
- Roofline Analysis

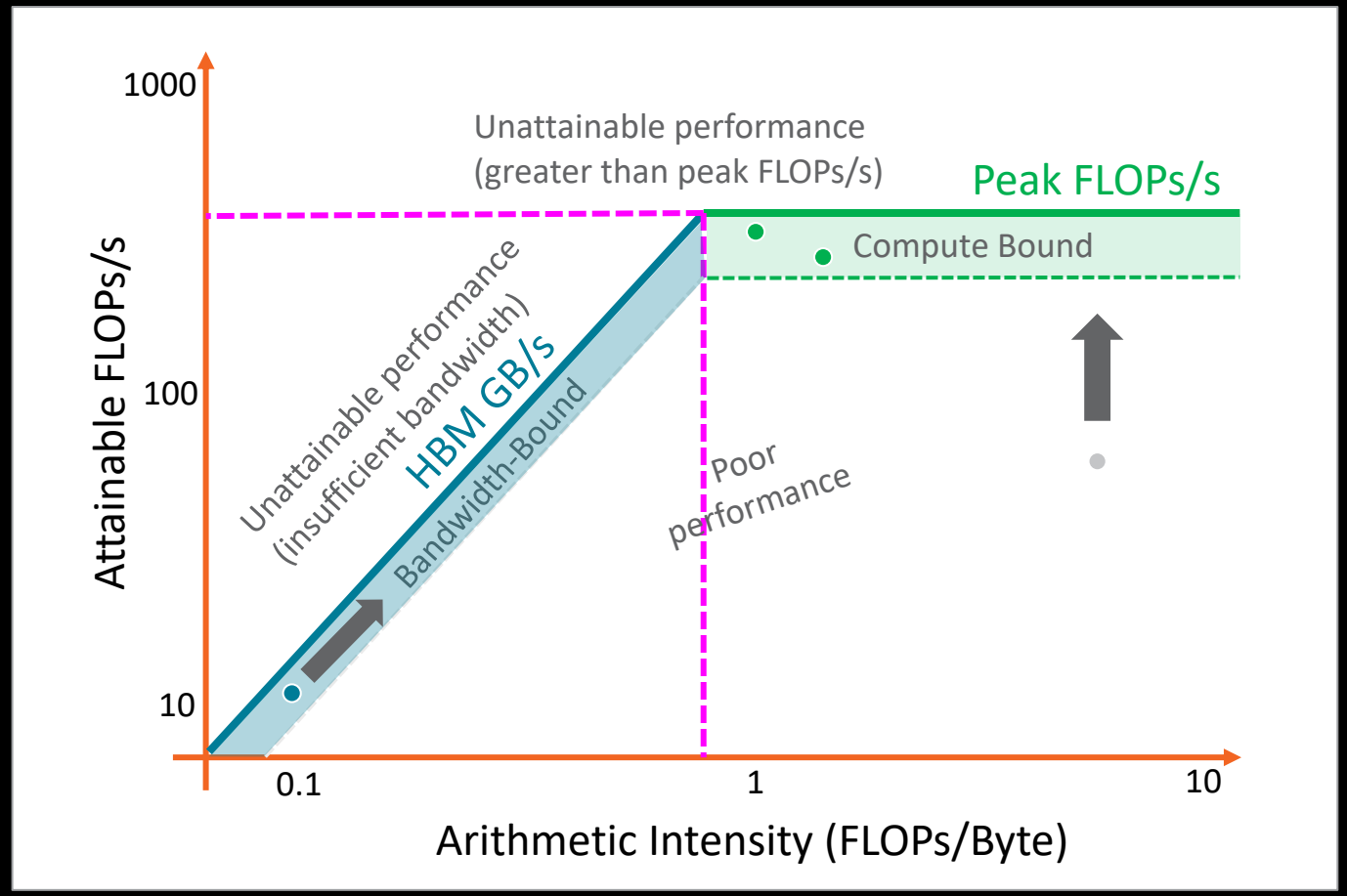


Derived Empirical Roofline analysis broken into two major instruction mixes. Showing application performance relative to measured maximum achievable performance. It uses the FLOP type selected while profiling

[Public]

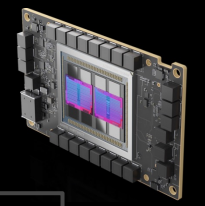
Background – What is roofline?

- 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



[Public]

Overview - AMD Instinct™ MI200 Architecture



Graphics Compute Die (GCD)

Compute Unit
Local Data Share (LDS)
Scalar Unit | SIMD0 | SIMD1 | SIMD2 | SIMD3
SGPR | VGPR | VGPR | VGPR | VGPR
Vector L1 Data Cache (vL1D)

Compute Unit
Local Data Share (LDS)
Scalar Unit | SIMD0 | SIMD1 | SIMD2 | SIMD3
SGPR | VGPR | VGPR | VGPR | VGPR
Vector L1 Data Cache (vL1D)

... ...

Compute Unit
Local Data Share (LDS)
Scalar Unit | SIMD0 | SIMD1 | SIMD2 | SIMD3
SGPR | VGPR | VGPR | VGPR | VGPR
Vector L1 Data Cache (vL1D)

L2 Cache (L2)

(Peer GCD)

Data Fabric

Remote Socket (CPU, GPU)

Memory Controller

(GCD1)

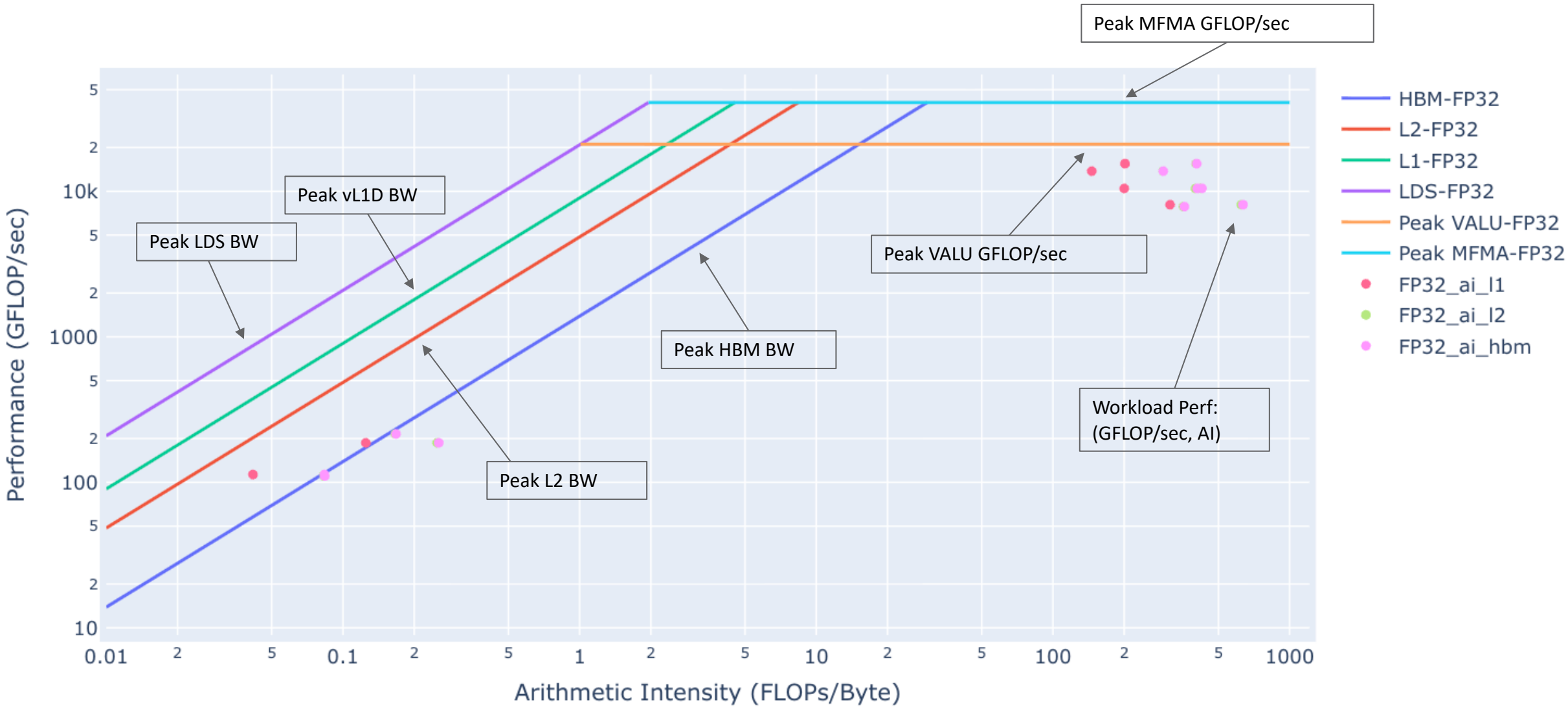
(GCD2)

HBM Memory

HBM Memory

[Public]

Empirical Hierarchical Roofline on MI200 - Overview



[Public]

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

```
10:57:35 amd@node-bp126-014a utils ±|master ✗|→ ./roofline
Total detected GPU devices: 2
GPU Device 0: Profiling...
 99% [||||||
```

[Public]

Low level Metrics

Section Title	Comments
Command Processor (CPC/CPF)	Packet processor data
Shader Processor Input (SPI)	Connecting packet processor and CUs
Wavefront Stats	Kernel launch stats
Compute Unit – Instruction Mix	Breakdown of instructions issued
Compute Unit – Compute Pipeline	
Texture Addressor & Texture Data (TA/TD)	Fetch & receive reqs for lookup in vL1D RAM
Local Data Share (LDS)	Cache level stats
Instruction Cache	
Scalar L1 Data Cache	
Vector L1 Data Cache	
L2 Cache	
L2 Cache (per channel)	

[Public]

Tips for Long-Running Benchmarks

- Filtering by kernel name and metrics during `Rocprof-compute profile` will cut down on profiling time
  - `Rocprof-compute profile –k “<kernel name>”` filters a single kernel name
  - `Rocprof-compute profile –k “<kernel1>” “<kernel2>”` filters two kernel names.
  - Note: surrounding a kernel name in quotes allows spaces to appear in your kernel search string
  - Also Note: Rocprof-compute applies the wildcard automatically, so only a unique substring of kernel names are required
  - Finally, Note: Rocprof-compute **`analyze`** `–k` does not take a kernel name, but an index
    - Index to kernel name mapping is given by `rocprof-compute analyze --list-stats`

- If you know which metrics you want to collect ahead of time, you can cut down how many rocprof runs are required
  - `Rocprof-compute profile --block SQ SQC –n <workload name> -- ./benchmark.sh` (SQC – Shader Sequencer Controller)
  - `Rocprof-compute profile --help` displays all block strings you can filter by
  - `Rocprof-compute` documentation: https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/conceptual/definitions.html goes over some of the meaning behind lower-level hardware units and metrics.



Example – DAXPY with a loop in the kernel

[Public]

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];
       }
}

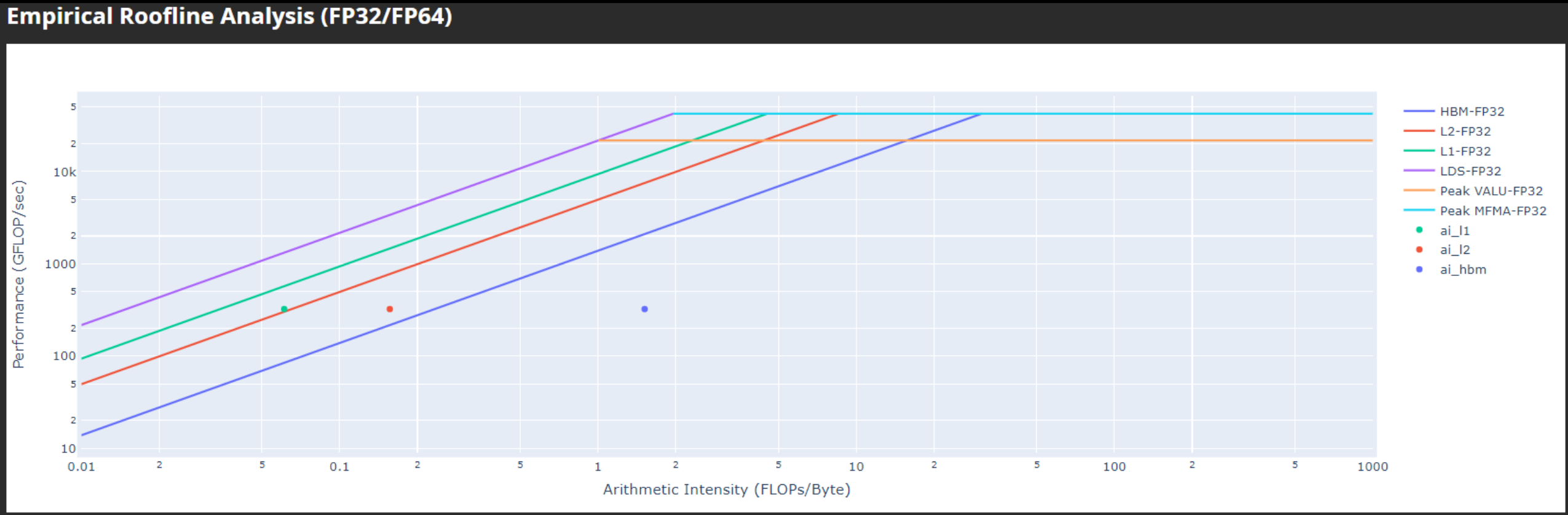
int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

    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();
```

[Public]

Roofline



- Performance: almost 330 GFLOPs

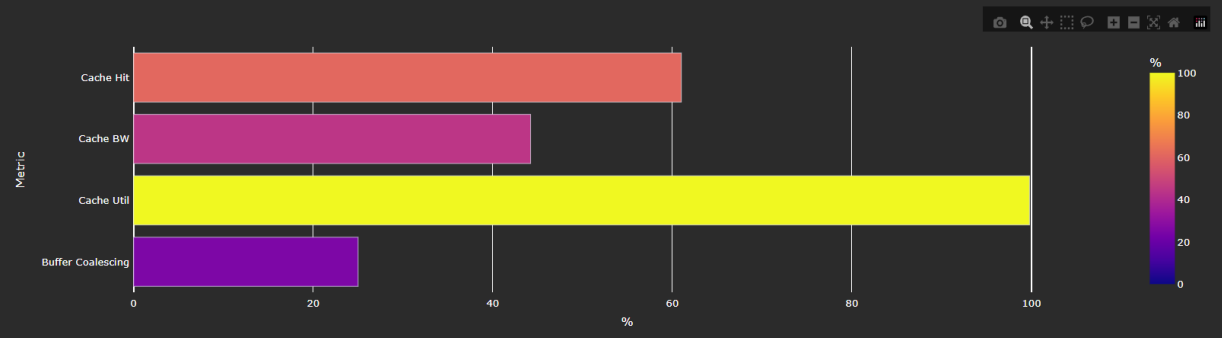
[Public]

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

[Public]

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];
       }
}

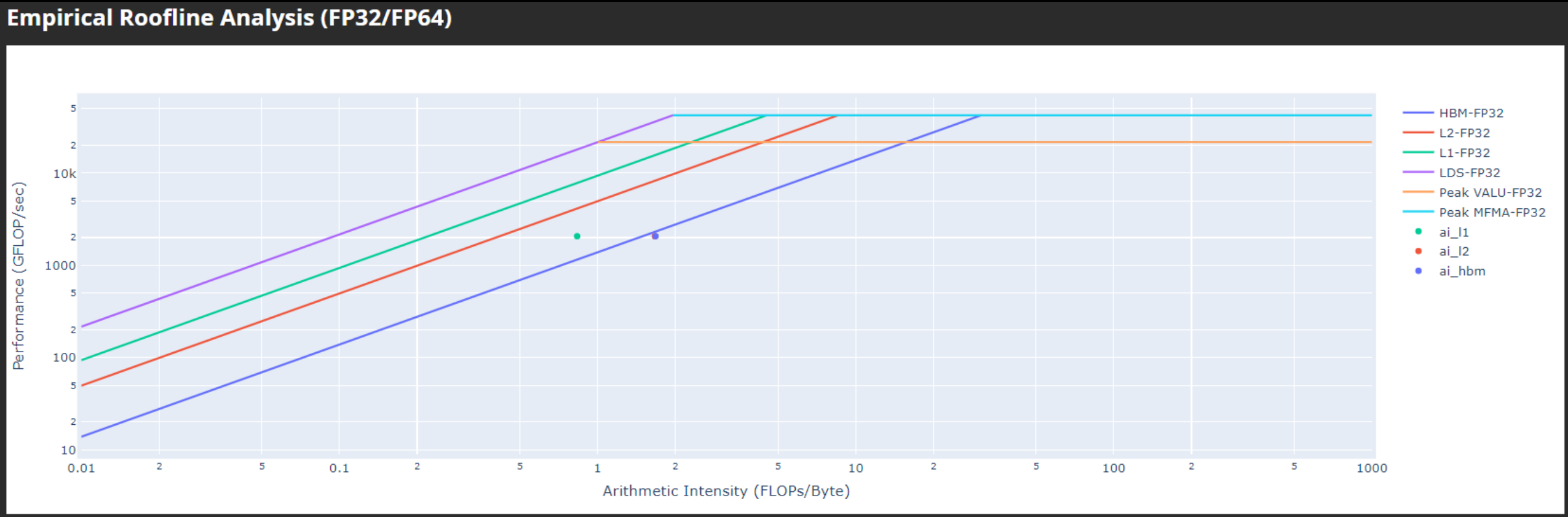
int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

    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();
```

[Public]

Roofline - Optimized



- Performance: almost 2 TFLOPs

AMD
together we advance_

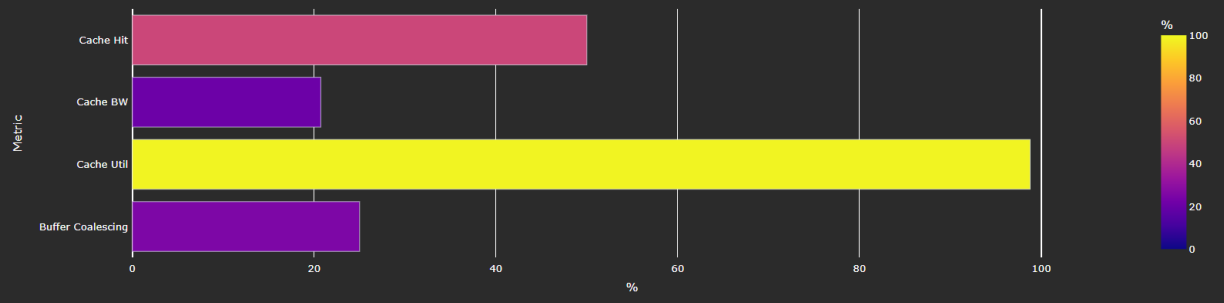
[Public]

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

[Public]

Another way: ROCprofv3 ATT and RCV

Visualization and analysis tool designed for GPU thread trace data collected using rocprofv3

OVERVIEW

ROCprof Compute Viewer (RCV) is a tool for visualizing and analyzing GPU advanced thread trace (ATT) data collected using rocprofv3.

PAIN POINTS

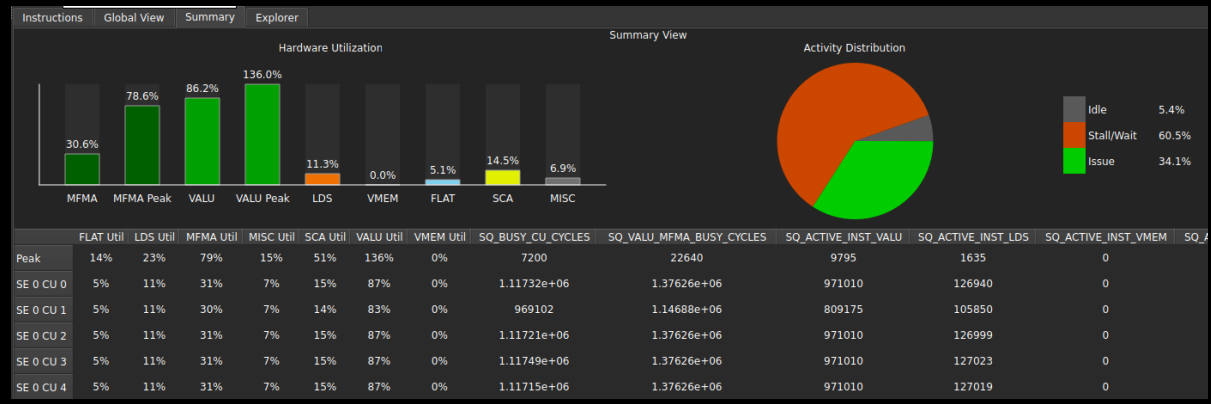
- What are the performance hotspots in my GPU kernels at the source and assembly levels?
- Which specific instructions and wavefronts cause performance bottlenecks?
- What is the hardware utilization over time?

KEY FEATURES

- Source Visualization (Trace -> assembly): Displays GPU thread traces, linking instructions to their corresponding source code and assembly for detailed execution mapping.
- Summary: Visualizes fast counter collection, load balancing information with hardware utilization by instruction type, per-compute unit utilization, peak rates, and compute unit/SIMD utilization.
- Compute Unit: Displays individual execution latency per wave (slot).
- Occupancy: Visualizes occupancy per Shader Engine (SE) and per kernel ID.
- Supports MI200, MI300, and MI350

https://rocm.docs.amd.com/projects/rocprof-compute-viewer/en/latest/

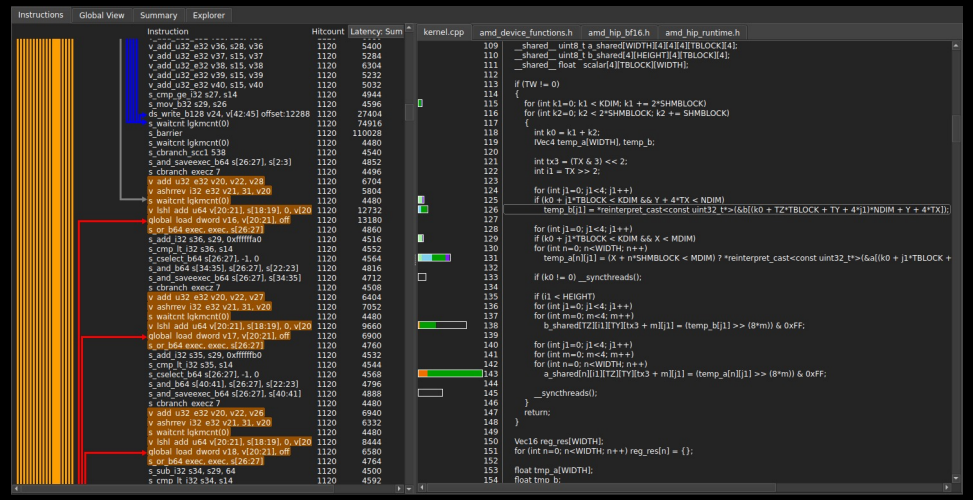
SUMMARY VIEW



	FLAT Util	LDS Util	MFMA Util	MISC Util	SCA Util	VALU Util	VMEM Util	SQ_BUSY_CU_CYCLES	SQ_VALU_MFMA_BUSY_CYCLES	SQ_ACTIVE_INST_VALU	SQ_ACTIVE_INST_LDS	SQ_ACTIVE_INST_VMEM
Peak	14%	23%	79%	15%	51%	136%	0%	7200	22640	9795	1635	0
SE 0 CU 0	5%	11%	31%	7%	15%	87%	0%	1.11732e+06	1.37626e+06	971010	126940	0
SE 0 CU 1	5%	11%	30%	7%	14%	83%	0%	969102	1.14688e+06	809175	105850	0
SE 0 CU 2	5%	11%	31%	7%	15%	87%	0%	1.11721e+06	1.37626e+06	971010	126999	0
SE 0 CU 3	5%	11%	31%	7%	15%	87%	0%	1.11749e+06	1.37626e+06	971010	127023	0
SE 0 CU 4	5%	11%	31%	7%	15%	87%	0%	1.11715e+06	1.37626e+06	971010	127019	0

Summary view shows results of application performance

INSTRUCTION VIEW



Instruction view is best suited for quickly finding hotspots in source

AMD
together we advance_

[Public]

Hands-on exercises

https://hackmd.io/@sfantao/lumi-training-cph-2026#Rocprof-compute

https://hackmd.io/@sfantao/lumi-training-cph-2026#Thread-tracing

We welcome you to explore our HPC Training Examples repo:

https://github.com/amd/HPCTrainingExamples

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

Relevant exercises for this presentation located in **rocprof-compute** directory.

Link to instructions on how to run the tests: **rocprof-compute/README.md and subdirectories**

[Public]

Questions?

[Public]

DISCLAIMERS AND ATTRIBUTIONS

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

© 2026 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, Radeon™, Instinct™, EPYC, Infinity Fabric, ROCm™, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.


AMD