AMD GPU Performance Analysis Tools

Presenter: Georgios Markomanolis
CINES - Hackathon
01/25/2024

AMD
together we advance_

Why Do We Need a Profiler?

Goals: better understand, characterize, and optimize any HPC/AI application while answering few questions including:

- Where the application spends most of the times
- How well the application is using the target device (CPU/GPU/ etc..)
- Why I am seeing this performance

This talk focuses on AMD GPU oriented profilers

AMD GPU Oriented Profilers

ROC-profiler (rocprof)

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

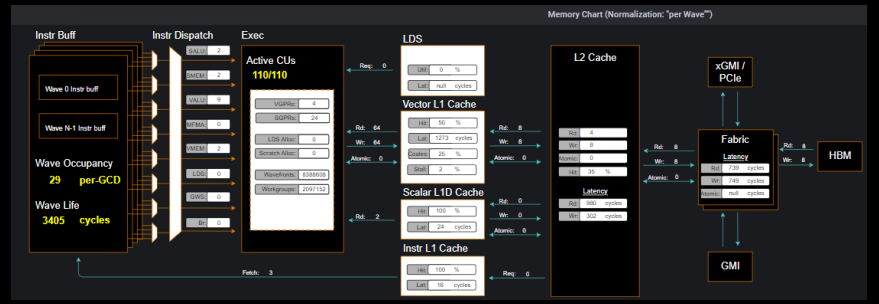
Omnitrace

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

Omniperf

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

	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



ROC-Profiler (rocprof)

What is ROC-Profiler?

- 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 `rocprof` can be visualized in the Chrome browser with Perfetto (https://ui.perfetto.dev/)

rocprof: Getting Started + Useful Flags

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

rocprof: Kernel Information

- rocprof can collect kernel(s) execution stats

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

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

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

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

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

rocprof: Collecting Application Traces

- rocprof can collect a variety of trace event types, and generate timelines in JSON format for use with Perfetto, currently:

Trace Event	rocprof Trace Mode
HIP API call	`--hip-trace`
GPU Kernels	`--hip-trace`
Host <-> Device Memory copies	`--hip-trace`
CPU HSA Calls	`--hsa-trace`
User code markers	`--roctx-trace`

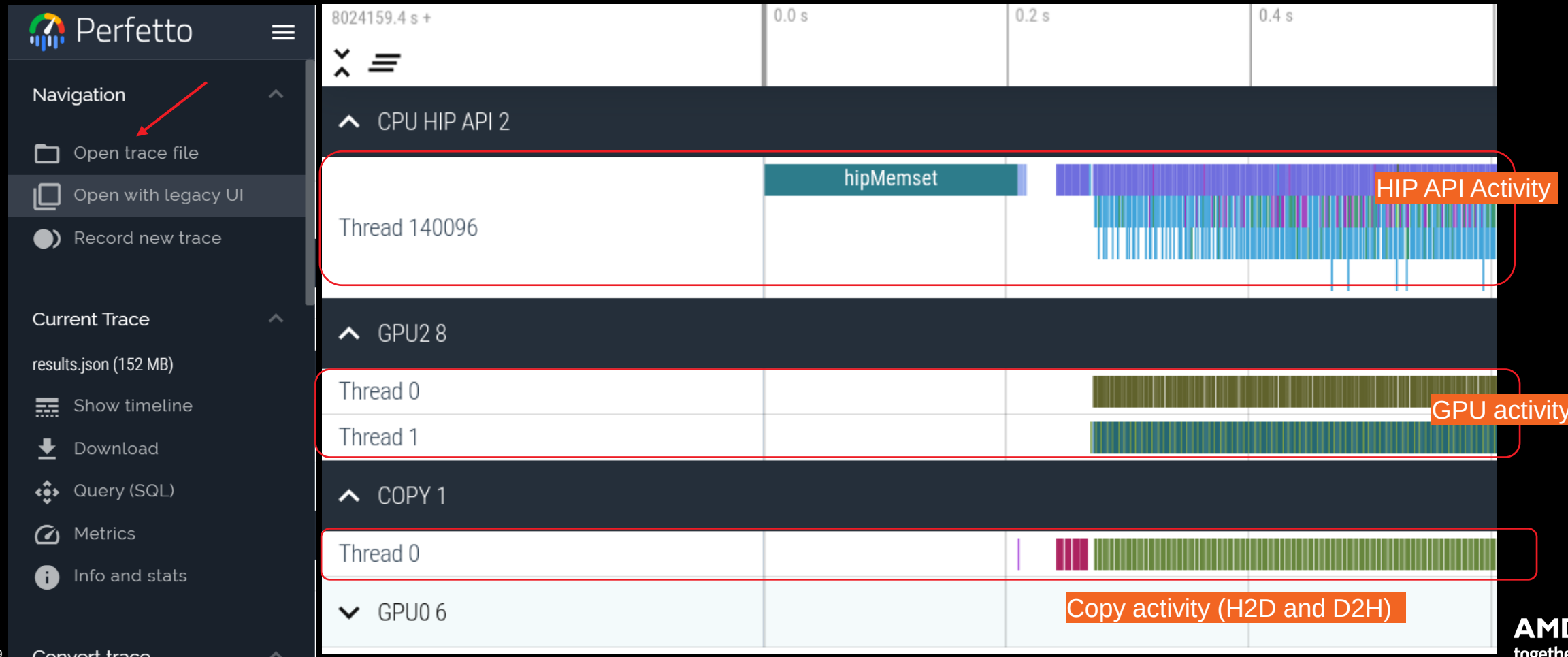
- You can combine modes like `--hip-trace --hsa-trace`
- If profiling OpenMP offload code, `--hsa-trace` is required to show HSA activity

rocprof + Perfetto: Collecting and Visualizing Application Traces

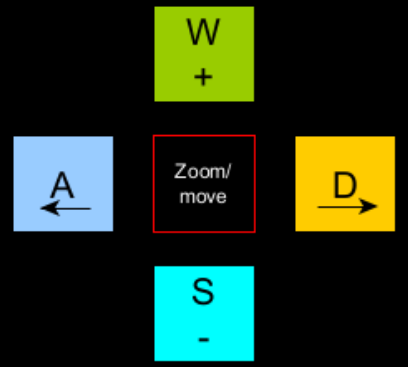
- rocprof can collect traces

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

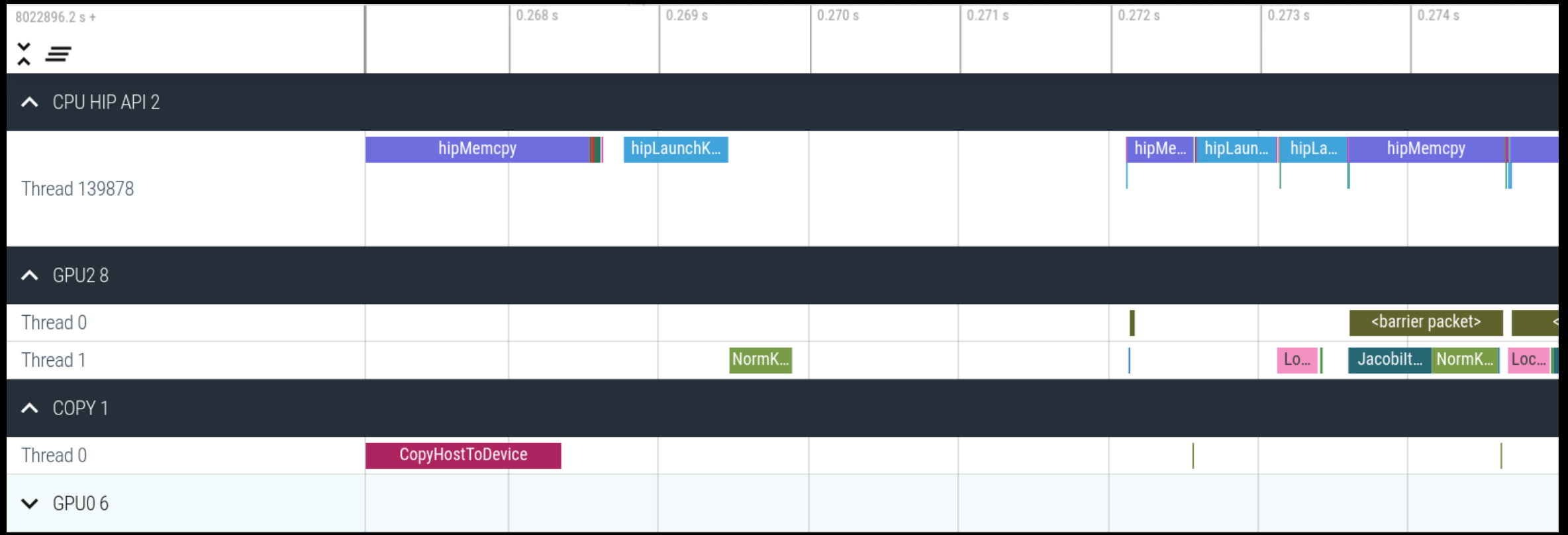
  This will output a .json file that can be visualized using the chrome browser and Perfetto (https://ui.perfetto.dev/)



Perfetto: Visualizing Application Traces

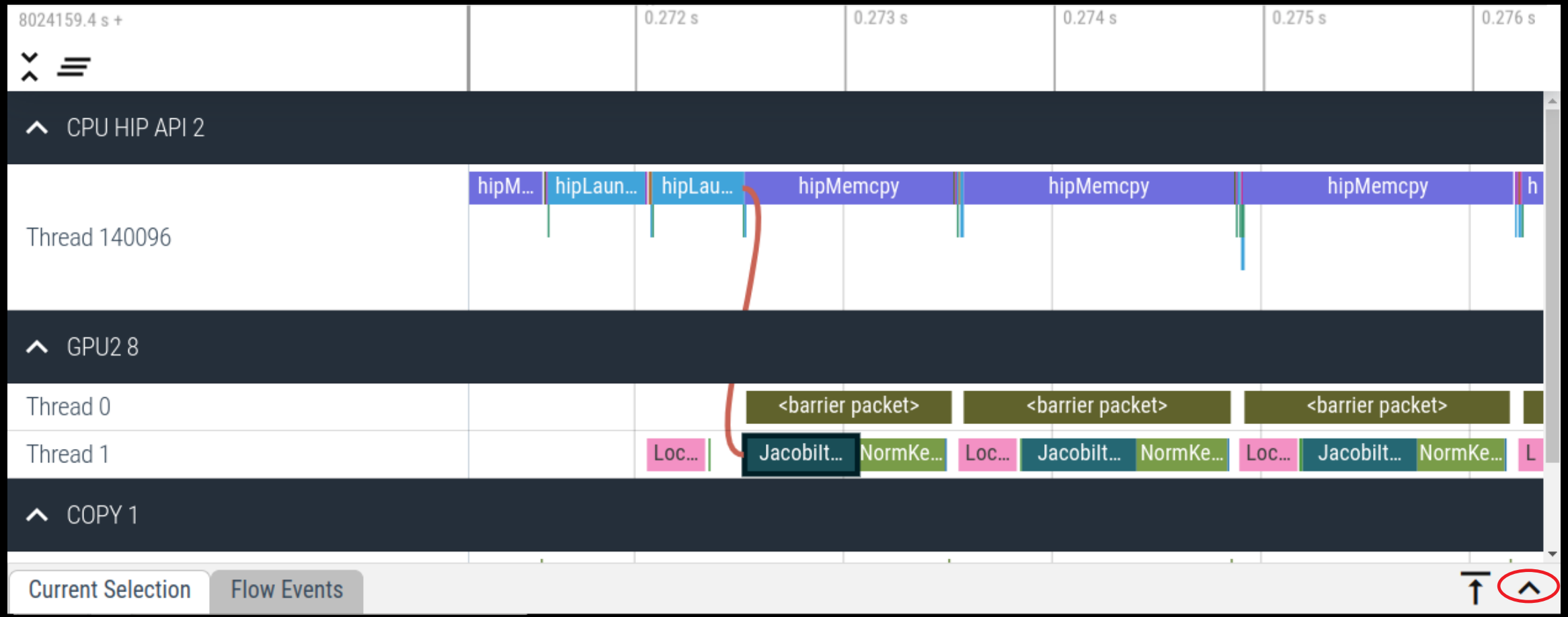


- Zoom in to see individual events
- Navigate trace using WASD keys

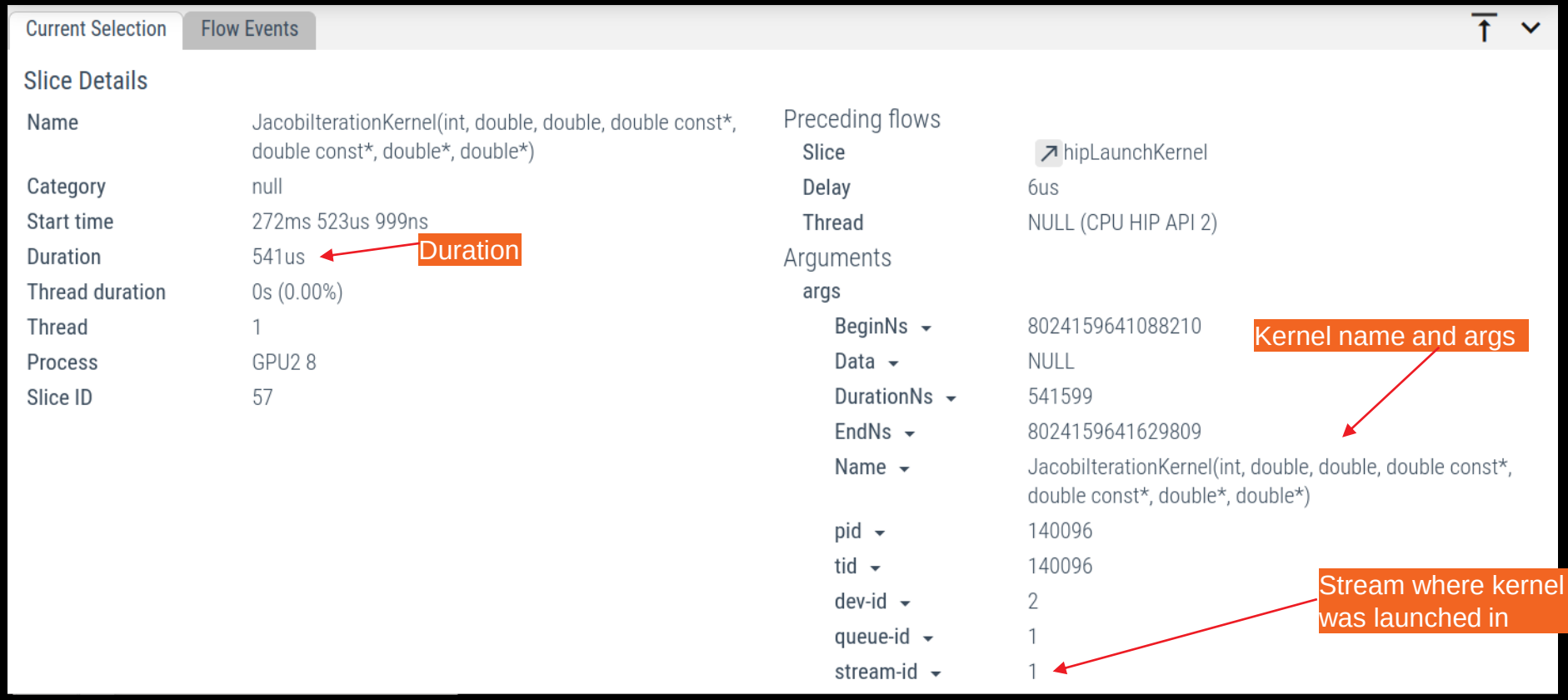


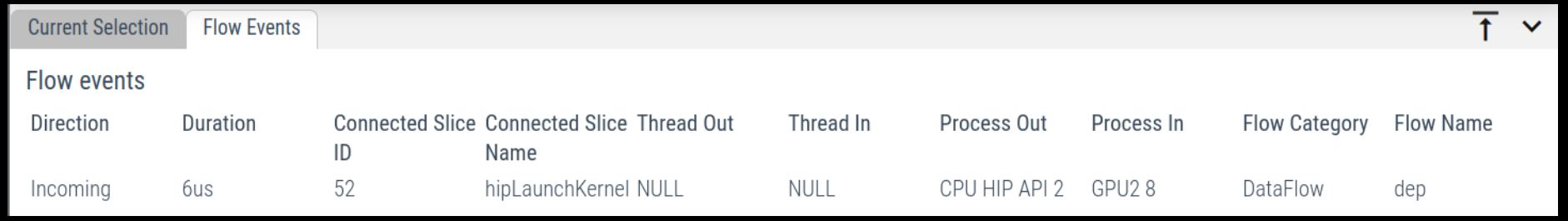
Perfetto: Kernel Information and Flow Events

- 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 and Flow Events





rocprof: Collecting Application Traces with rocTX Markers and Regions

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

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

- Annotate code with roctx markers:

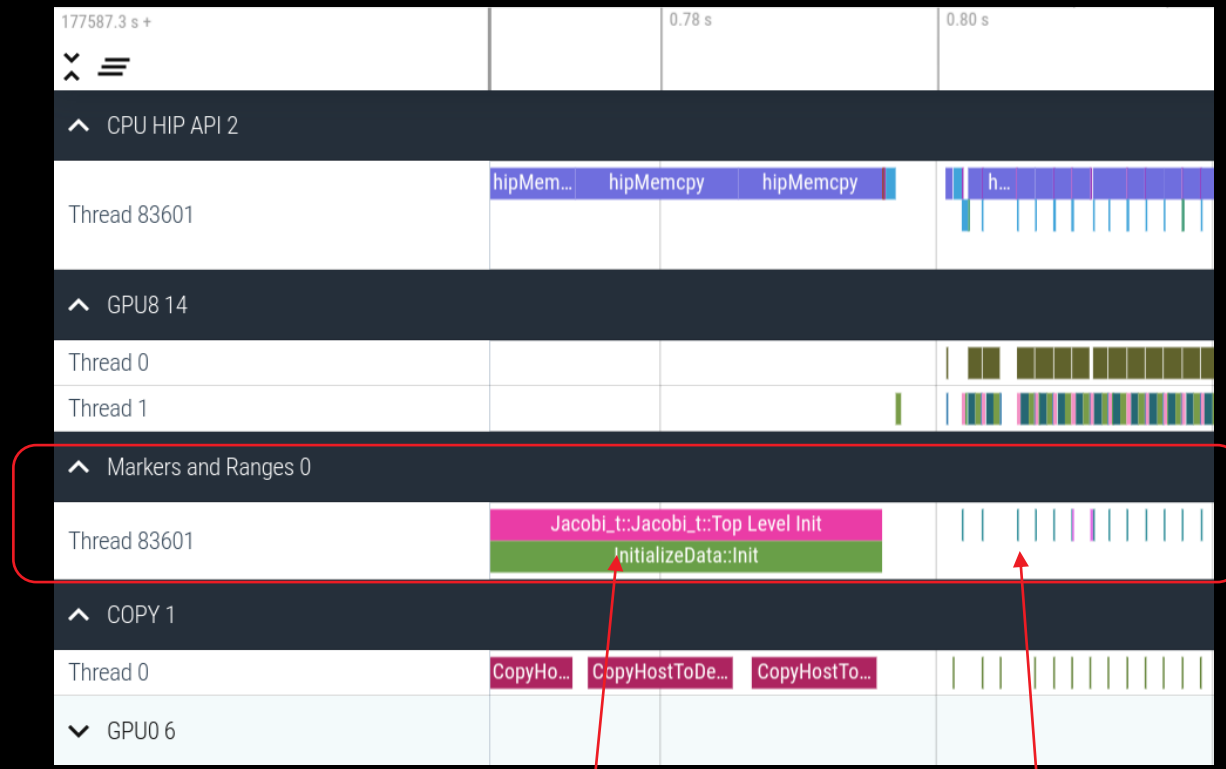
```
...
    roctxMark("start of some code");
    // some_code
    roctxMark("end of some code");
...
```

- Add roctx and roctracer libraries to link line:

```
-L${ROCM_PATH}/lib -lroctx64 -lroctracer64
```

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

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



rocprof: Collecting Hardware Counters

- rocprof can collect a number of hardware counters and derived counters
  - `$ /opt/rocm/bin/rocprof --list-basic`
  - `$ /opt/rocm/bin/rocprof --list-derived`

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

```
pmc : Wavefronts VALUInsts VFetchInsts VWriteInsts VALUUtilization VALUBusy WriteSize
pmc : SALUInsts SFetchInsts LDSInsts FlatLDSInsts GDSInsts SALUBusy FetchSize
pmc : L2CacheHit MemUnitBusy MemUnitStalled WriteUnitStalled ALUStalledByLDS LDSBankConflict
```

  - 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

  - A csv file will be created containing all the requested counters for each invocation of every kernel

rocprof: 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

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

- How do you time a kernel's duration?
  - `$ /opt/rocm/bin/rocprof --timestamp on -i rocprof_counters.txt <app with args>`
  - This produces four times: `DispatchNs, BeginNs, EndNs,` and `CompleteNs`
  - Closest thing to a kernel duration: `EndNs - BeginNs`
  - If you run with "`--stats`" the resultant results.stats.csv file will include a kernel duration column
    - Note: the duration is aggregated over repeated calls to the same kernel

rocprof: Multiple MPI Ranks

- rocprof can collect counters and traces for multiple MPI ranks
- Say you want to profile an application usually called like this:

```
mpiexec –np <n> ./Jacobi_hip –g <x> <y>
```

- Invoke the profiler by executing:

```
mpiexec -np <n> rocprof <rocprof_options> ./Jacobi_hip -g <x> <y>
or
srun –-ntasks=n rocprof <rocprof_options> ./Jacobi_hip -g <x> <y>
```

- By directing output files from each rank to different directories, we can collect traces for each rank separately
  - Use a helper script for this, and run your program as shown below:

```
mpiexec -np <n> helper_rocprof.sh ./Jacobi_hip -g <x> <y>
```

- Multi-node profiling currently isn’t supported

Profiling Per MPI Rank As sbtach Job(1)

- Let's consider a 3-step run:
  - `sbatch_profiling.sh` with sbatch command line to launch the app
  - `rocprof_batch.slurm` This file contains sbatch parameters and the call to srun command line
  - `rocprof_wrapper.sh` calls rocprof command line with input parameters to run the application to be profiled

- `$ cat sbatch_profiling.sh`

```
sbatch -p <partition> -w <node> rocprof_batch.slurm
```

- `$cat rocprof_batch.slurm`

```
#!/bin/bash
#SBATCH --job-name=run
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=2
#SBATCH --gpus-per-task=1
#SBATCH --cpus-per-task=1
#SBATCH --distribution=block:block
#SBATCH --time=00:20:00
#SBATCH --output=out.txt
#SBATCH --error=err.txt
#SBATCH -A XXXXX
cd ${SLURM_SUBMIT_DIR}

#load necessary modules
#export necessary environment variables

make clean all
srun ./rocprof_wrapper.sh ${repository} triad_off_mpi triad_off_mpi
```

Profiling Per MPI Rank As sbatch Job(2)

$cat rocprof_wrapper.sh

```
#!/bin/bash
set -euo pipefail
# depends on ROCM_PATH being set outside;  input arguments are the output directory & the name
outdir="$1"
name="$2"
if [[ -n ${OMPI_COMM_WORLD_RANK+z} ]]; then
    # mpich
    export MPI_RANK=${OMPI_COMM_WORLD_RANK}
elif [[ -n ${MV2_COMM_WORLD_RANK+z} ]]; then
    # ompi
    export MPI_RANK=${MV2_COMM_WORLD_RANK}
elif [[ -n ${SLURM_PROCID+z} ]]; then
    export MPI_RANK=${SLURM_PROCID}
else
    echo "Unknown MPI layer detected! Must use OpenMPI, MVAPICH, or SLURM"
    exit 1
fi
rocprof="${ROCM_PATH}/bin/rocprof"

pid="$$"
outdir="${outdir}/rank_${pid}_${MPI_RANK}"
outfile="${name}_${pid}_${MPI_RANK}.csv"
${rocprof} -d ${outdir} --hsa-trace -o ${outdir}/${outfile} "${@:3}"
```

Output directory per rank

Filenames annotated with rank as well

Application and its arguments

rocprof: Profiling Overhead

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

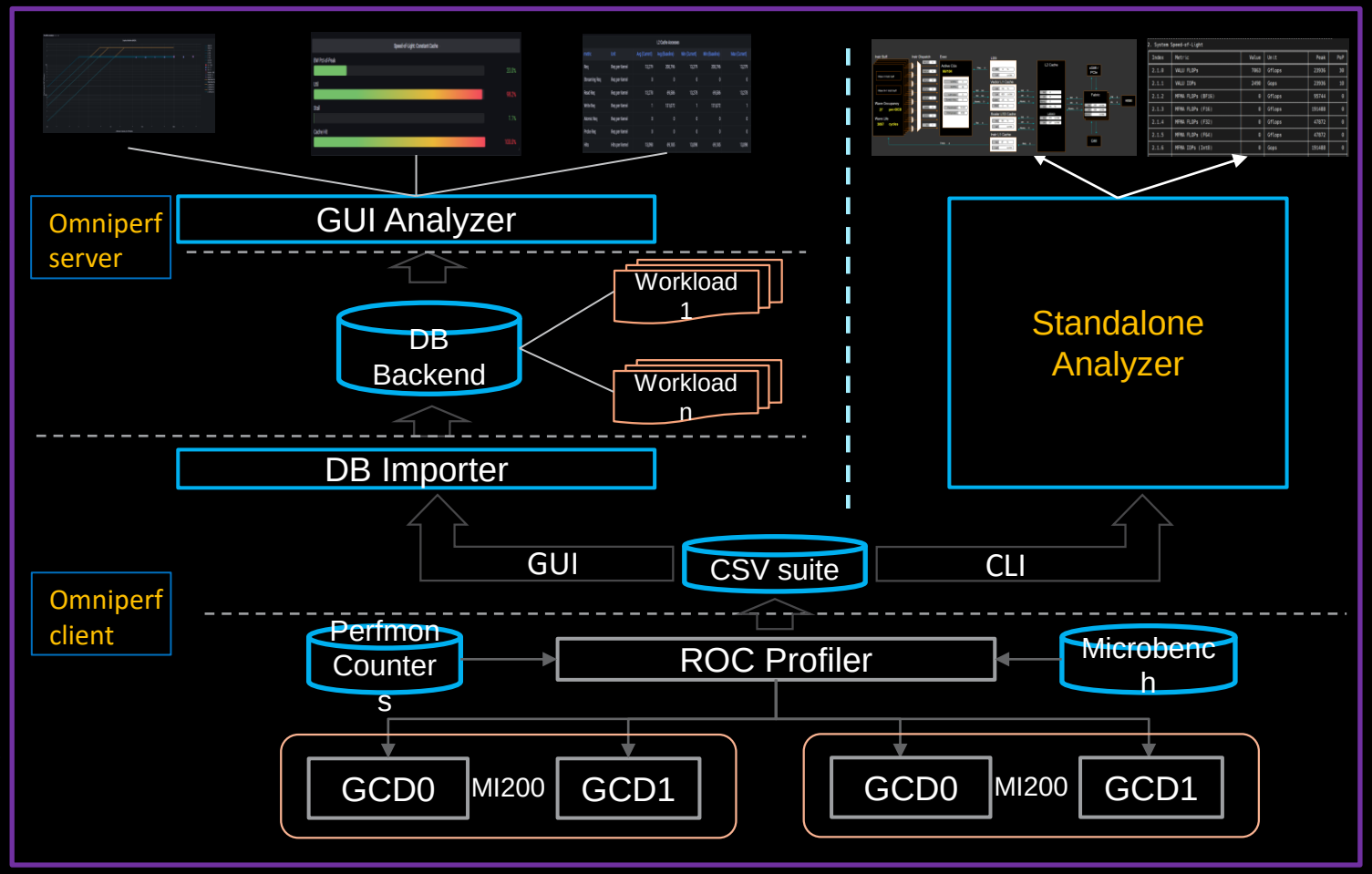
rocprof: Summary

- rocprof is the open source, command line AMD GPU profiling tool distributed with ROCm
- Many other tools are built over rocprof
- rocprof provides tracing of GPU kernels, HIP API, HSA API and Copy activity
- rocprof can be used to collect GPU hardware counters with additional overhead
- JSON Traces can be viewed in Perfetto UI
- Other output files are in text/CSV format

Omniperf

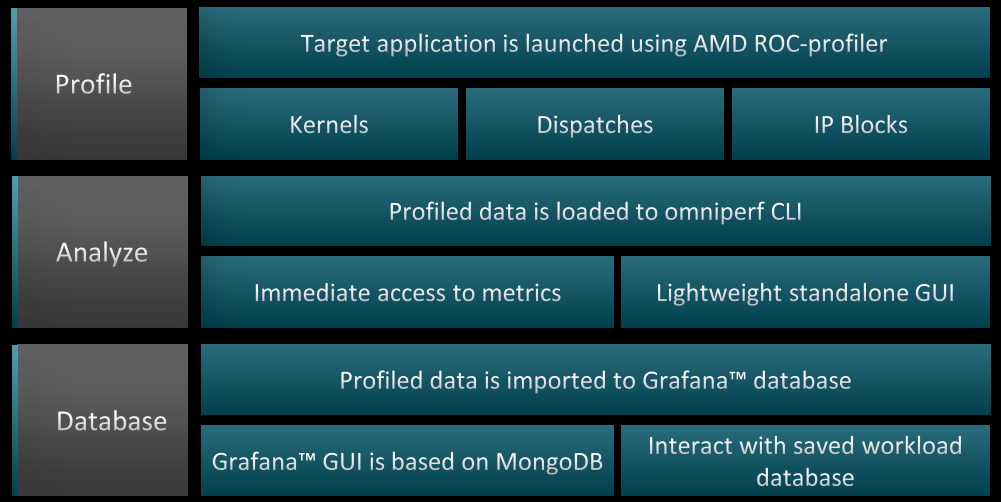
Omniperf: Automated Collection of Hardware Counters and Analysis

AMD Research Tool	Repository: https://github.com/AMDResearch/omniperf			
	✗ Not part of ROCm stack	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

Omniperf modes



Basic command-line syntax:

Profile:

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

Analyze:

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

To use a lightweight standalone GUI with CLI analyzer:

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

Database:

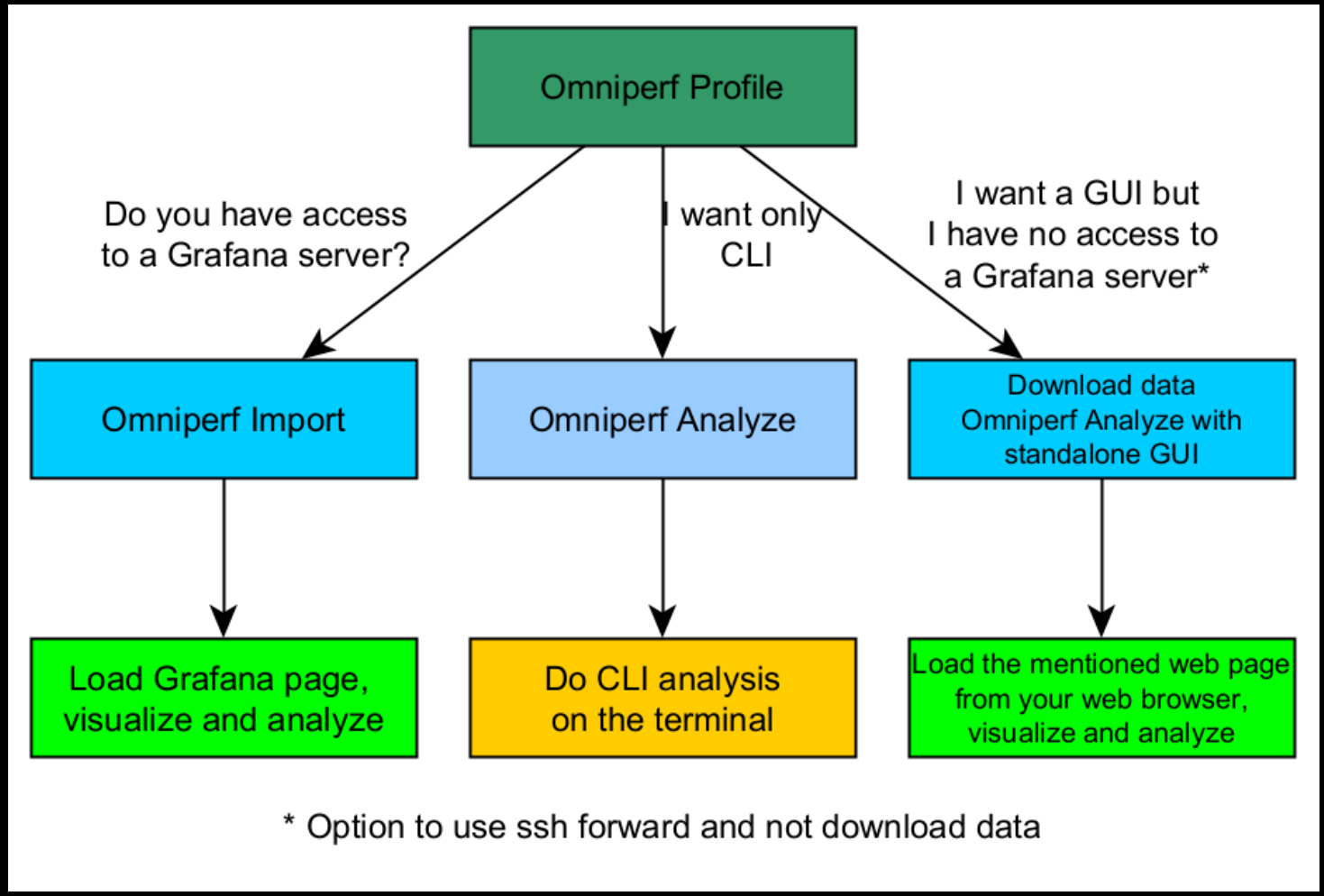
```
$ omniperf database <interaction type> [connection options]
```

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

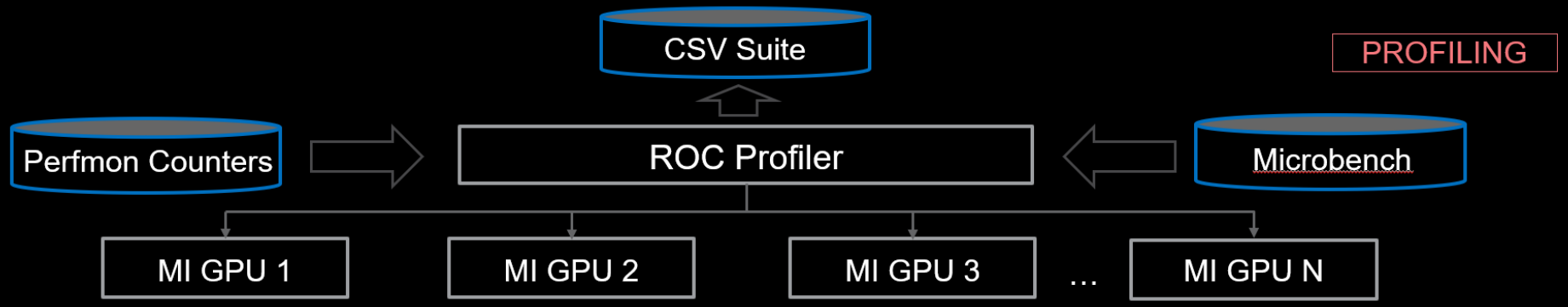
```
$ omniperf profile --help
```

For problems, create an issue here: https://github.com/AMDResearch/omniperf/issues
Documentation: https://amdresearch.github.io/omniperf

Omniperf workflows



Profile Mode

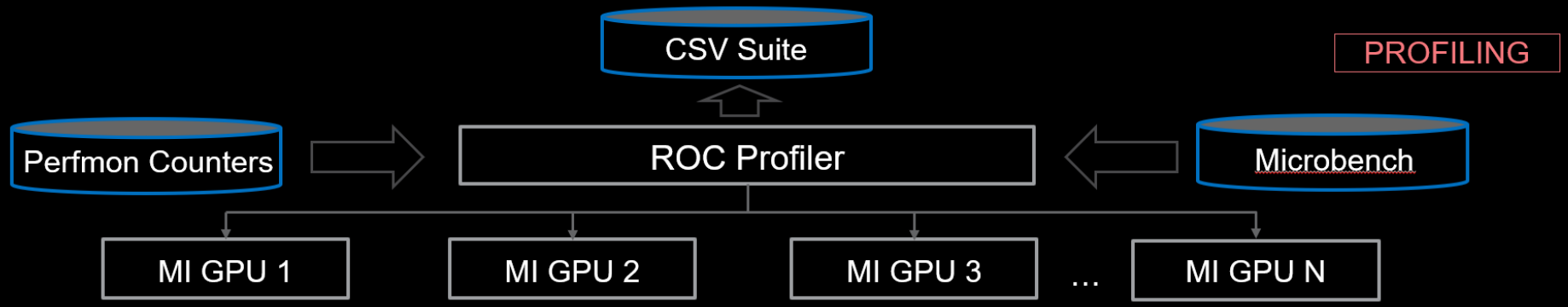


Features:

- Runtime Filtering
  `--kernel`, `--ipblocks`, `--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.

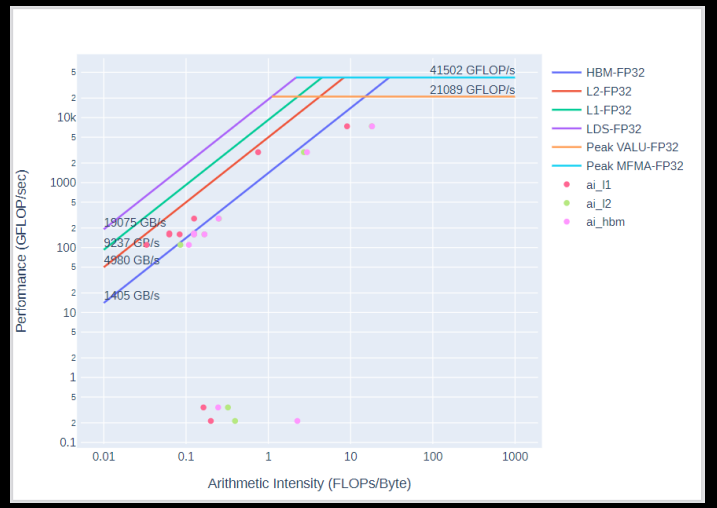
Profile Mode



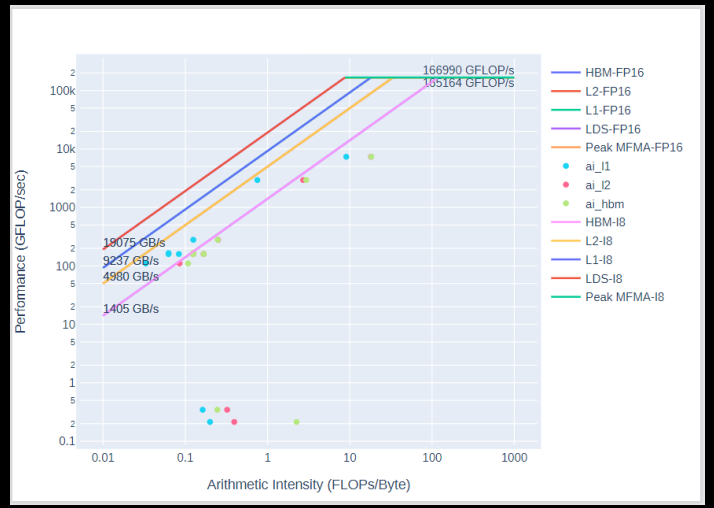
Features:

- Runtime Filtering
  `--kernel`, `--ipblocks`, `--dispatch`
- Standalone Roofline Analysis
  `--roof-only`

FP32/FP64

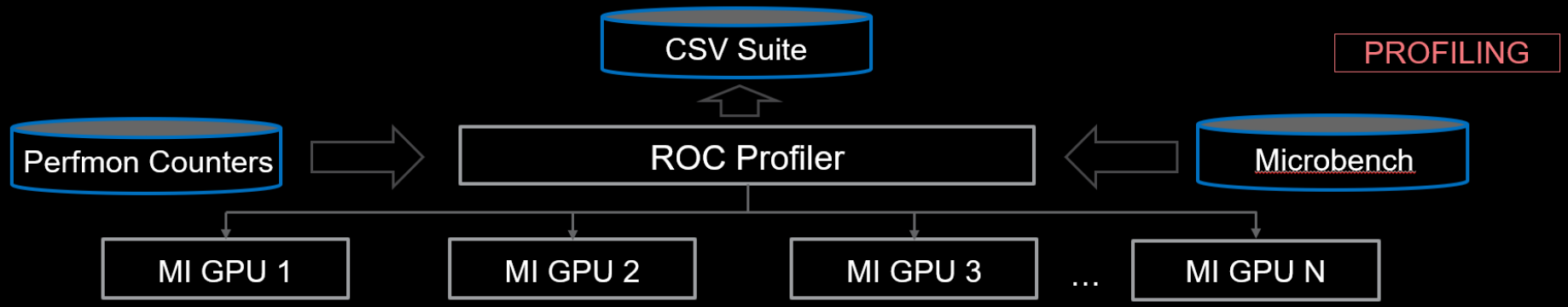


FP16/INT8



The above plots are saved as PDF output when the --roof-only option is used

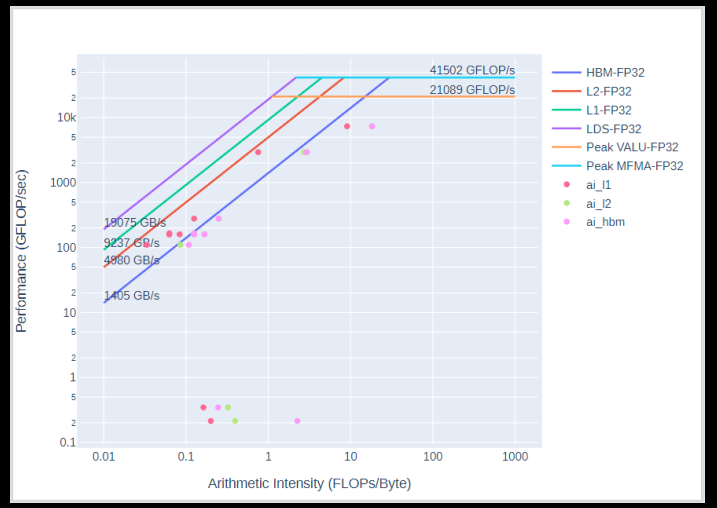
Profile Mode



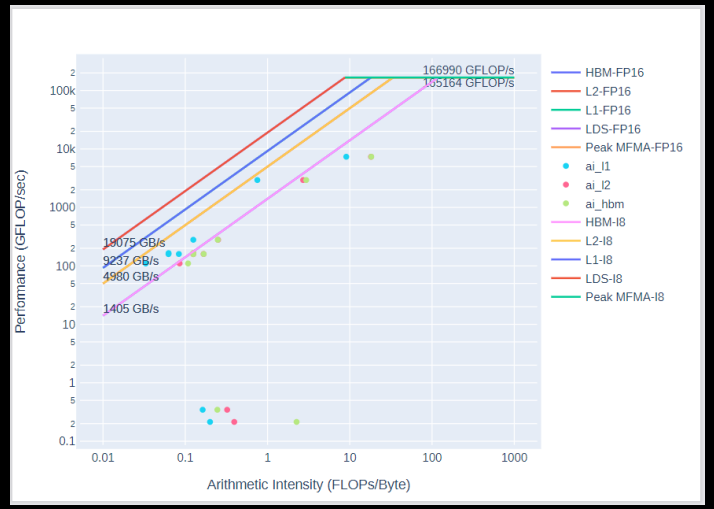
Features:

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

FP32/FP64



FP16/INT8



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

Omniperf profiling: Example

We use the example sample/vcopy.cpp from the Omniperf installation folder:

```
$ wget https://github.com/AMDResearch/omniperf/raw/main/sample/vcopy.cpp
```

Compile with hipcc:

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

Profile with Omniperf:

```
$ omniperf profile –n vcopy_all -- ./vcopy 1048576 256
…
--------------
Profile only
--------------

omniperf ver:  1.0.4
Path:  /pfs/lustrep4/scratch/project_462000075/markoman/omniperf-
1.0.4/build/workloads
Target:  mi200
Command:  ./vcopy 1048576 256
Kernel Selection:  None
Dispatch Selection:  None
IP Blocks: All
```

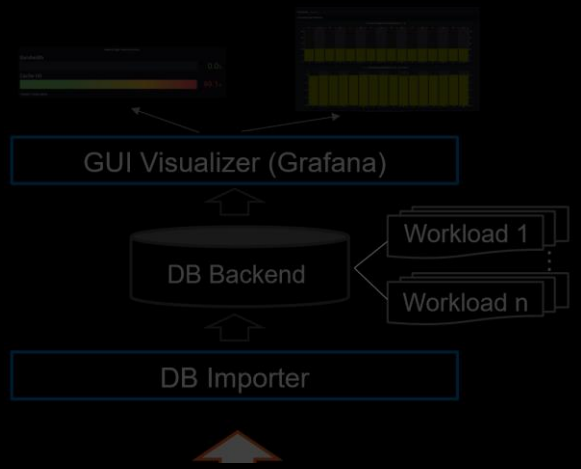
A new directory will be created called `workloads/vcopy_all`

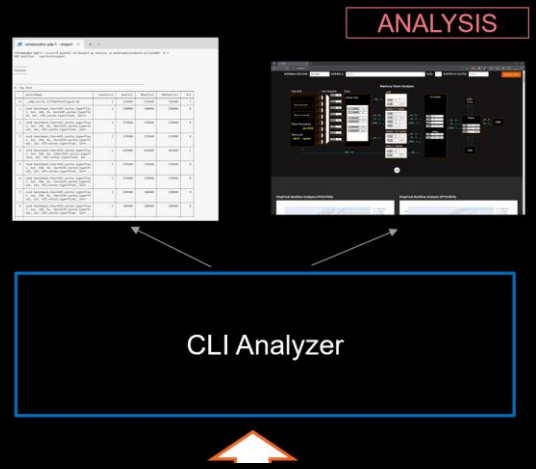
For help:

```
$ omniperf profile –h
```

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

Analyze Mode





Features:

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

```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --list-kernels

--------
Analyze
--------

--------------------------------------------------------------------------------
Detected Kernels
```

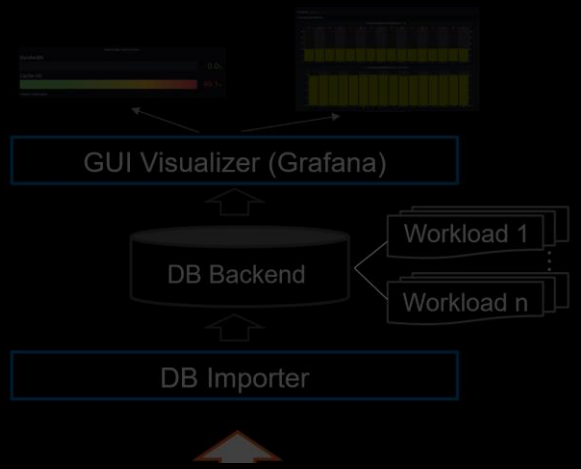
	KernelName
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vecto
2	void benchmark_func<double, 256, 8u, 512u>(double, double*) [clone .kd]
3	void benchmark_func<int, 256, 8u, 256u>(int, int*) [clone .kd]
4	void benchmark_func<__half2, 256, 8u, 512u>(__half2, __half2*) [clone .kd]
5	void benchmark_func<float, 256, 8u, 512u>(float, float*) [clone .kd]
6	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 256u>(HIP_vector_type<float, 2u>, HIP_vecto
7	void benchmark_func<double, 256, 8u, 256u>(double, double*) [clone .kd]
8	void benchmark_func<int, 256, 8u, 128u>(int, int*) [clone .kd]
9	void benchmark_func<__half2, 256, 8u, 256u>(__half2, __half2*) [clone .kd]

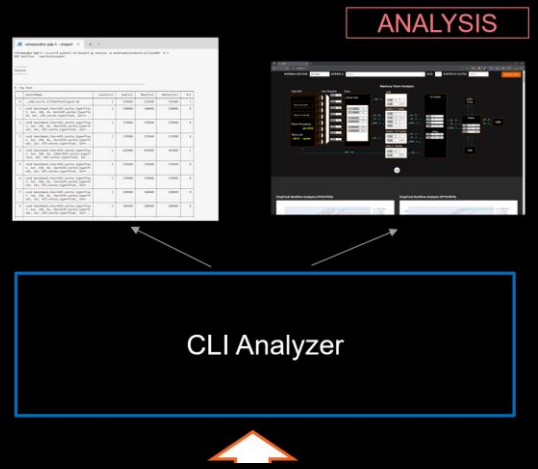
```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --list-metrics gfx90a
```

	Metric
0	Top Stat
1	System Info
2.1.0	VALU_FLOPs
2.1.1	VALU_IOPs
2.1.2	MFMA_FLOPs_(BF16)
2.1.3	MFMA_FLOPs_(F16)
2.1.4	MFMA_FLOPs_(F32)
2.1.5	MFMA_FLOPs_(F64)
2.1.6	MFMA_IOPs_(Int8)
2.1.7	Active_CUs
2.1.8	SALU_Util
2.1.9	VALU_Util
2.1.10	MFMA_Util
2.1.11	VALU_Active_Threads/Wave
2.1.12	IPC - Issue

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

Analyze Mode





Features:

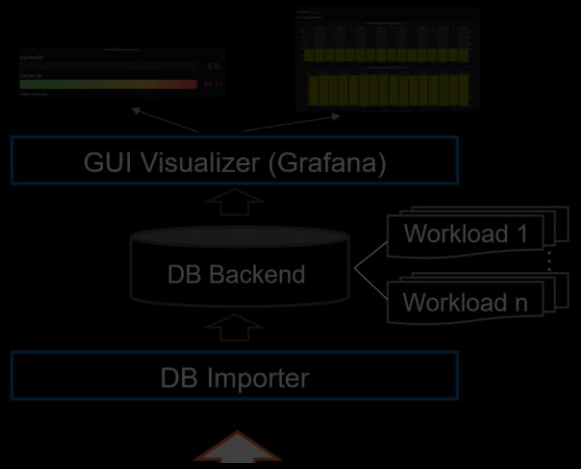
- List top kernels or view list of metrics
  `--list-kernels`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`

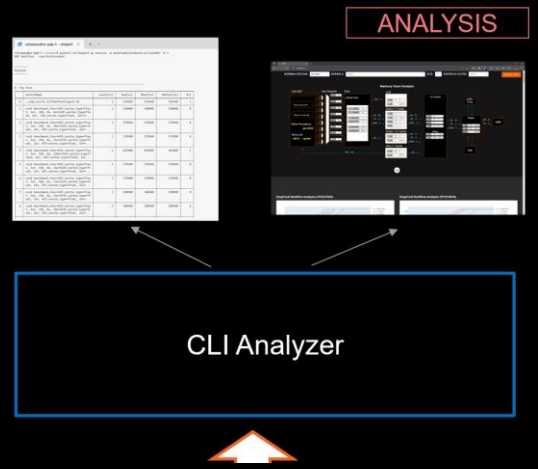
```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --kernel 0

--------
Analyze
--------

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

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct	S
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]	1	3353042.00	3353042.00	3353042.00	7.87	*
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vector_type<float, 2u>...	1	1721239.00	1721239.00	1721239.00	4.04	
2	void benchmark_func<double, 256, 8u, 512u>(double, double*) [clone .kd]	1	1710840.00	1710840.00	1710840.00	4.02	
3	void benchmark_func<int, 256, 8u, 256u>(int, int*) [clone .kd]	1	1693880.00	1693880.00	1693880.00	3.98	
4	void benchmark_func<__half2, 256, 8u, 512u>(__half2, __half2*) [clone .kd]	1	1670521.00	1670521.00	1670521.00	3.92	
5	void benchmark_func<float, 256, 8u, 512u	1	1661402.00	1661402.00	1661402.00	3.90	

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

Analyze Mode





Features:

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

```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --metric 5

--------
Analyze
--------

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

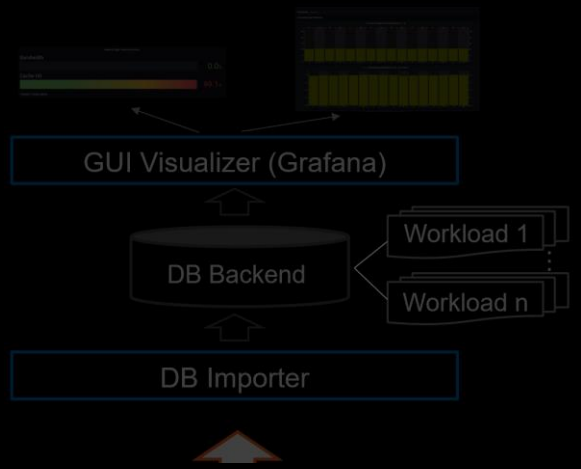
	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]	1	3353042.00	3353042.00	3353042.00	7.87
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vector_type<float, 2u>...	1	1721239.00	1721239.00	1721239.00	4.04
2	void benchmark_func<double, 256, 8u, 512u>(double, double*) [clone .kd]	1	1710840.00	1710840.00	1710840.00	4.02
3	void benchmark_func<int, 256, 8u, 256u>(int, int*) [clone .kd]	1	1693880.00	1693880.00	1693880.00	3.98
4	void benchmark_func<__half2, 256, 8u, 512u>(__half2, __half2*) [clone .kd]	1	1670521.00	1670521.00	1670521.00	3.92
5	void benchmark_func<float, 256, 8u, 512u>(float, float*) [clone .kd]	1	1661402.00	1661402.00	1661402.00	3.90
6	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 256u>(HIP_vector_type<float, 2u>, HIP_vector_type<float, 2u>...	1	881739.00	881739.00	881739.00	2.07
7	void benchmark_func<double, 256, 8u, 256u>(double, double*) [clone .kd]	1	875980.00	875980.00	875980.00	2.06
8	void benchmark_func<int, 256, 8u, 128u>(int, int*) [clone .kd]	1	865100.00	865100.00	865100.00	2.03
9	void benchmark_func<__half2, 256, 8u, 256u>(__half2, __half2*) [clone .kd]	1	855660.00	855660.00	855660.00	2.01

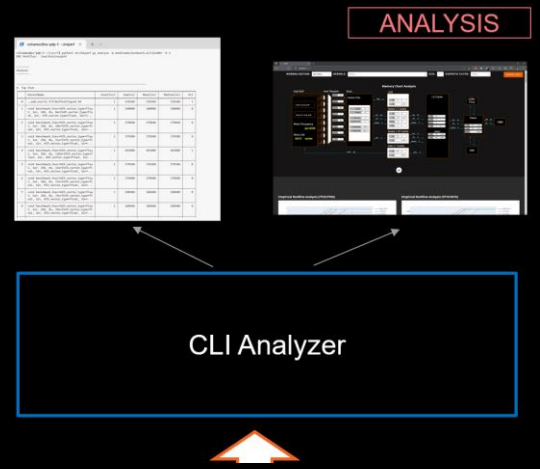
```
--------------------------------------------------------------------------------
5. Command Processor (CPC/CPF)
5.1 Command Processor Fetcher
```

Index	Metric	Avg	Min	Max	Unit
5.1.0	GPU Busy Cycles	416535.02	29084.00	5253061.00	Cycles/kernel
5.1.1	CPF Busy	416535.02	29084.00	5253061.00	Cycles/kernel

Filtering output to isolate data table at index 5

Analyze Mode





Features:

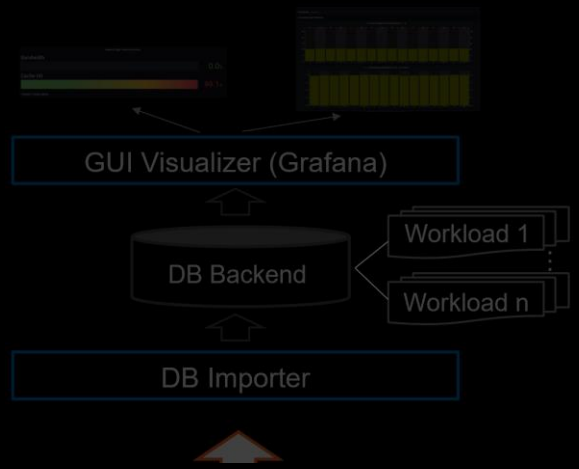
- List top kernels or view list of metrics
  `--list-kernels`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`
- Filter by metric id(s)
  `--metric`
- Change normalization unit, time unit, or decimal
  `--normal-unit`, `--time-unit`, `--decimal`

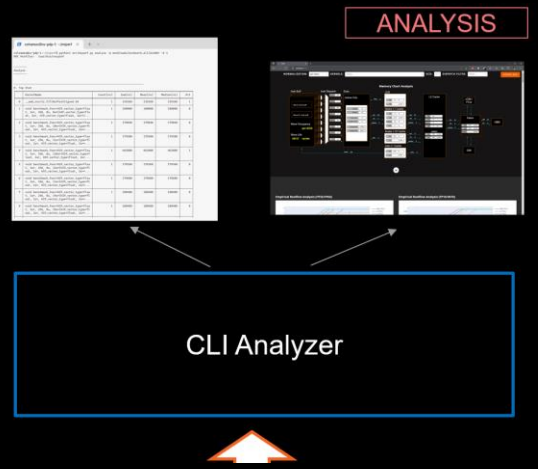
7.2 Wavefront Runtime Stats

Index	Metric	Avg	Min	Max	Unit
7.2.0	Kernel Time (Nanosec)	255131.78	8480.00	3353042.00	Ns
7.2.1	Kernel Time (Cycles)	416535.02	29084.00	5253061.00	Cycle
7.2.2	Instr/wavefront	557.11	48.00	9300.00	Instr/wavefront
7.2.3	Wave Cycles	18777.13	1848.52	258296.68	Cycles per wave
7.2.4	Dependency Wait Cycles	2819.92	942.73	10169.97	Cycles per wave
7.2.5	Issue Wait Cycles	14105.31	100.13	211703.70	Cycles per wave
7.2.6	Active Cycles	2161.27	180.00	36172.00	Cycles per wave
7.2.7	Wavefront Occupancy	2770.80	185.11	3224.96	Wavefronts

Output showing the default normalization and time unit

Analyze Mode (cont.)





Features:

- Baseline Analysis

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

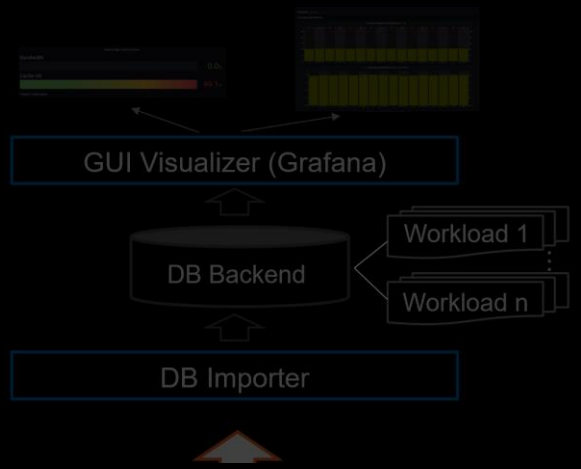
2. System Speed-of-Light

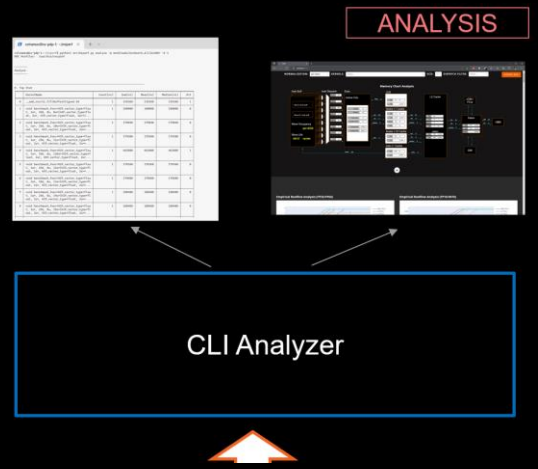
Index	Metric	Value	Value	Unit	Peak	Peak	PoP	PoP
2.1.0	VALU FLOPs	7492.7178288728755	0.0 (-100.0%)	GFlop	22630.4	22630.4 (0.0%)	33.10908260071795	0.0 (-100.0%)
2.1.1	VALU IOPs	2326.1937250093497	398.91 (-82.85%)	Giop	22630.4	22630.4 (0.0%)	10.279065880449968	1.76 (-82.85%)
2.1.2	MFMA FLOPs (BF16)	0.0	0.0 (nan%)	GFlop	90521.6	90521.6 (0.0%)	0.0	0.0 (nan%)
2.1.3	MFMA FLOPs (F16)	0.0	0.0 (nan%)	GFlop	181043.2	181043.2 (0.0%)	0.0	0.0 (nan%)
2.1.4	MFMA FLOPs (F32)	0.0	0.0 (nan%)	GFlop	45260.8	45260.8 (0.0%)	0.0	0.0 (nan%)
2.1.5	MFMA FLOPs (F64)	0.0	0.0 (nan%)	GFlop	45260.8	45260.8 (0.0%)	0.0	0.0 (nan%)
2.1.6	MFMA IOPs (Int8)	0.0	0.0 (nan%)	Giop	181043.2	181043.2 (0.0%)	0.0	0.0 (nan%)
2.1.7	Active CUs	102	74.0 (-27.45%)	Cus	104	104.0 (0.0%)	98.07692307692308	71.15 (-27.45%)
2.1.8	SALU Util	2.6093901009614555	3.62 (38.57%)	Pct	100	100.0 (0.0%)	2.6093901009614555	3.62 (38.57%)
2.1.9	VALU Util	58.371669678115765	5.17 (-91.15%)	Pct	100	100.0 (0.0%)	58.371669678115765	5.17 (-91.15%)
2.1.10	MFMA Util	0.0	0.0 (nan%)	Pct	100	100.0 (0.0%)	0.0	0.0 (nan%)
2.1.11	VALU Active Threads/Wave	64.0	64.0 (0.0%)	Threads	64	64.0 (0.0%)	100.0	100.0 (0.0%)
2.1.12	IPC - Issue	1.0	1.0 (0.0%)	Instr/cycle	5	5.0 (0.0%)	20.0	20.0 (0.0%)
2.1.13	LDS BW	0.0	0.0 (nan%)	Gb/sec	22630.4	22630.4 (0.0%)	0.0	0.0 (nan%)
2.1.14	LDS Bank Conflict		0.0 (nan%)	Conflicts/access	32	32.0 (0.0%)		0.0 (nan%)
2.1.15	Instr Cache Hit Rate	99.99239800871251	99.91 (-0.08%)	Pct	100	100.0 (0.0%)	99.99239800871251	99.91 (-0.08%)
2.1.16	Instr Cache BW	1687.4579645653916	227.95 (-86.49%)	Gb/s	6092.8	6092.8 (0.0%)	27.695935605393114	3.74 (-86.49%)
2.1.17	Scalar L1D Cache Hit Rate	99.34855885851496	99.82 (0.47%)	Pct	100	100.0 (0.0%)	99.34855885851496	99.82 (0.47%)
2.1.18	Scalar L1D Cache BW	57.584644049561916	227.95 (295.85%)	Gb/s	6092.8	6092.8 (0.0%)	0.9451261168848792	3.74 (295.85%)
2.1.19	Vector L1D Cache Hit Rate	20.359281437125 75	50.0 (145.59%)	Pct	100	100.0 (0.0%)	20.359281437125 75	50.0 (145.59%)
2.1.20	Vector L1D Cache BW	1699.7181220813884	1823.61 (7.29%)	Gb/s	11315.199999999999	11315.2 (0.0%)	15.021547317602769	16.12 (7.29%)
2.1.21	L2 Cache Hit Rate	3.8149067110455 04	35.21 (822.95%)	Pct	100	100.0 (0.0%)	3.8149067110455 04	35.21 (822.95%)
2.1.22	L2-Fabric Read BW	1166.9922392326407	456.37 (-60.89%)	Gb/s	1638.4	1638.4 (0.0%)	71.2275536641016	27.85 (-60.89%)
2.1.23	L2-Fabric Write BW	6.623892610383628	320.42 (4737.3%)	Gb/s	1638.4	1638.4 (0.0%)	0.4042903204579851	19.56 (4737.3%)
2.1.24	L2-Fabric Read Latency	536.7282175696066	282.93 (-47.29%)	Cycles		0.0 (nan%)		0.0 (nan%)
2.1.25	L2-Fabric Write Latency	401.33373490590895	332.3 (-17.2%)	Cycles		0.0 (nan%)		0.0 (nan%)
2.1.26	Wave Occupancy	2770.796874555133	1848.05 (-33.3%)	Wavefronts	3328	3328.0 (0.0%)	83.25711762485373	55.53 (-33.3%)
2.1.27	Instr Fetch BW	405.02278909507197	0.0 (-100.0%)	Gb/s	3046.4	3046.4 (0.0%)	13.295128318509454	0.0 (-100.0%)
2.1.28	Instr Fetch Latency	18.298147264262635	21.37 (16.76%)	Cycles		0.0 (nan%)		0.0 (nan%)

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

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit

Analyze Mode (cont.)





Features:

- Baseline Analysis

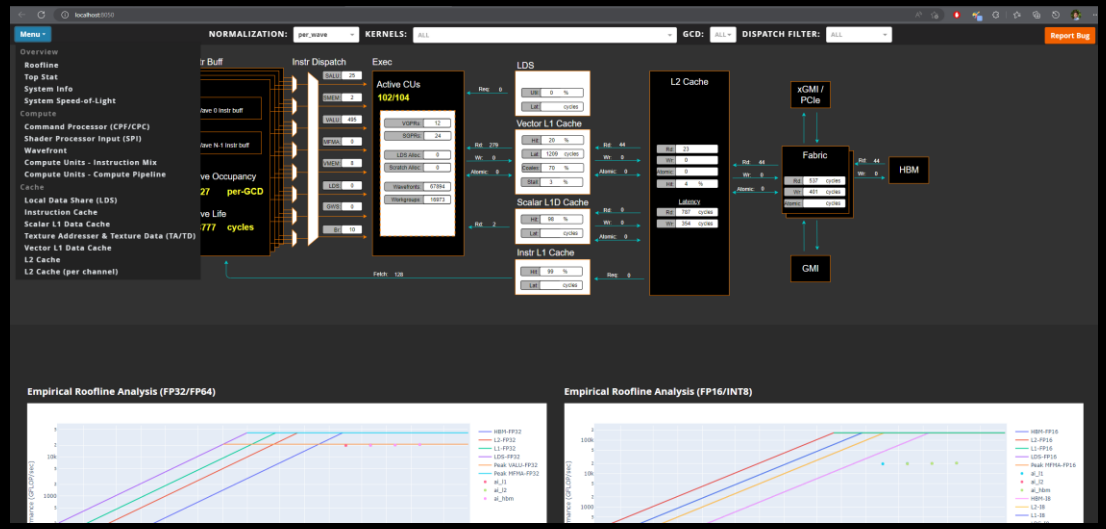
  `--path <workload1_path> --path <workload2_path>`

- Launch a standalone HTML page from terminal

  `--gui <port>`

```
colramos@sv-pdp-2:~$ omniperf analyze -p workloads/mix_all/mi200/ --gui

--------
Analyze
--------

Dash is running on http://0.0.0.0:8050/

 * Serving Flask app 'omniperf_analyze.omniperf_analyze'
 * Debug mode: off
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://127.0.0.1:8050
 * Running on http://10.228.33.182:8050
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

Omniperf analyze: Example

We use the example sample/vcopy.cpp from the Omniperf installation folder:

```
$ wget https://github.com/AMDResearch/omniperf/raw/main/sample/vcopy.cpp
```

Compile with hipcc:

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

Profile with Omniperf:

```
$ omniperf profile -n vcopy_all -- ./vcopy 1048576 256
```

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

Analyze the profiled workload:

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

For help:

```
$ omniperf analyze -h
```

0. Top Stat

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
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	0.08	Instr/cycle	5	10.576640831930312

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

Omniperf analyze: Example with Standalone GUI Usage

We use the example sample/vcopy.cpp from the Omniperf installation folder:

```
$ wget https://github.com/AMDResearch/omniperf/raw/main/sample/vcopy.cpp
```

Compile with hipcc:

```
$ hipcc –o vcopy vcopy.cpp
```

Profile with Omniperf:

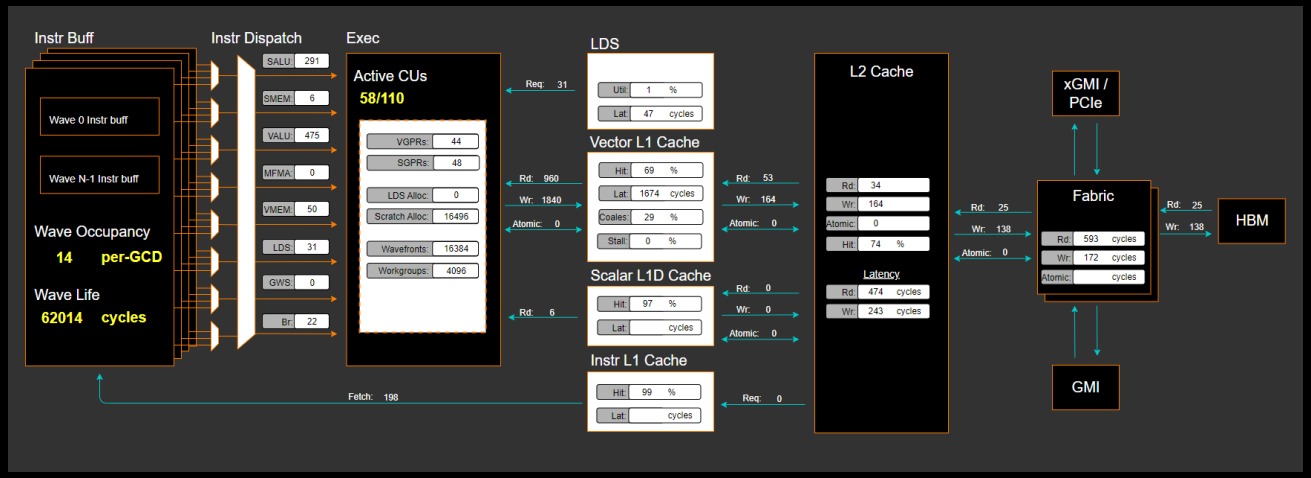
```
$ omniperf profile –n vcopy_all -- ./vcopy 1048576 256
```

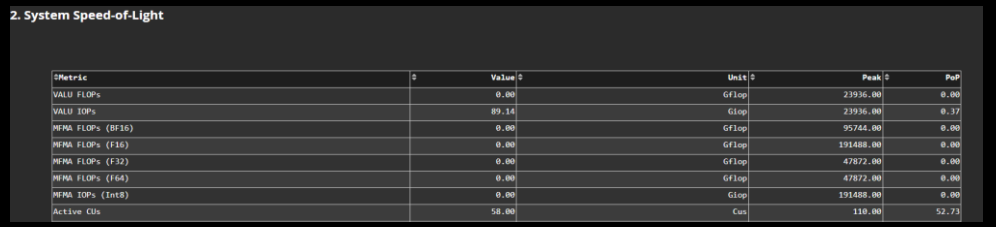
A new directory will be created called `workloads/vcopy_all`

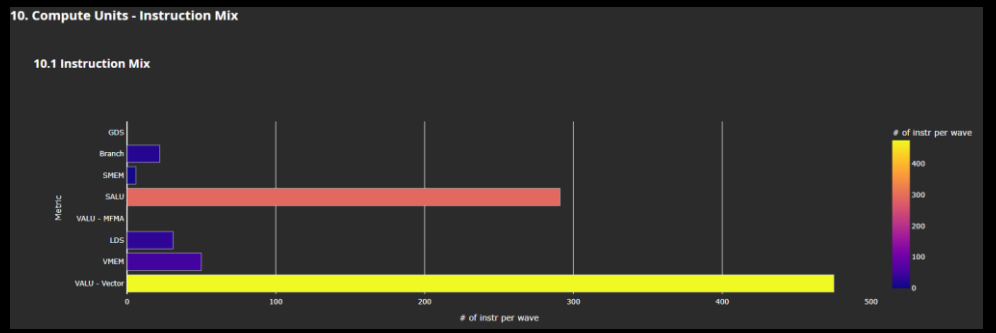
Analyze the profiled workload:

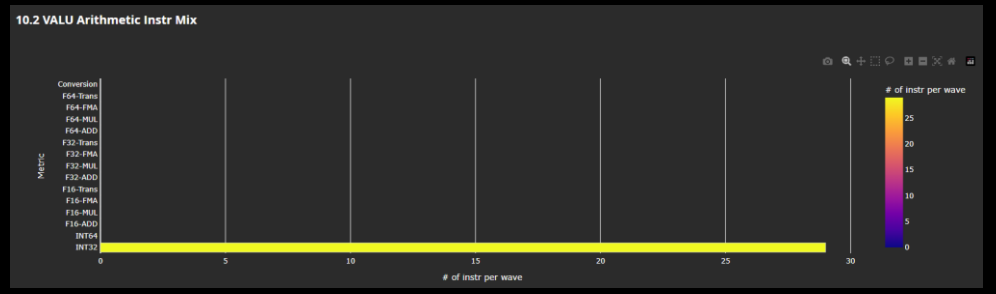
```
$ omniperf analyze –p workloads/vcopy_all/mi200/ --gui
```

Open web page http://IP:8050/









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 Omniperf 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

- Omniperf profile has an option called –k where you define which specific kernel to profile.
- 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 omniperf profile -n kernel_roof
-k kernel_name --roof-only  -- ./binary args
```

Guided Exercises

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

We will discuss only a few of them here

Guided Exercises: Logistics/Preamble

- To accommodate the virtual setting and attendees with varied access to Omniperf:
  - I'll read through the slides without waiting for everyone to finish working through each exercise
  - If you have access to a system with Omniperf, clone the repo and start working through the exercises:
    - git clone https://github.com/amd/HPCTrainingExamples/, we'll be working in the OmniperfExamples/1-LaunchParameters subdirectory.
    - The READMEs contain all of what I'm saying and include platform-specific instructions for this training in the top-level directory

- We have used a publicly available release candidate of Omniperf to generate output for these slides:
  - https://github.com/AMDResearch/omniperf/releases/tag/v1.1.0-PR1
  - Behavior may differ if using a different version of Omniperf (e.g. 1.0.10)
  - Generally, building stable releases is the best practice

- The numbers shown in the READMEs and these slides were generated using MI210 accelerators

- 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:
  - Omniperf commands to filter through output for common optimization concerns
  - Some optimization direction given certain Omniperf 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

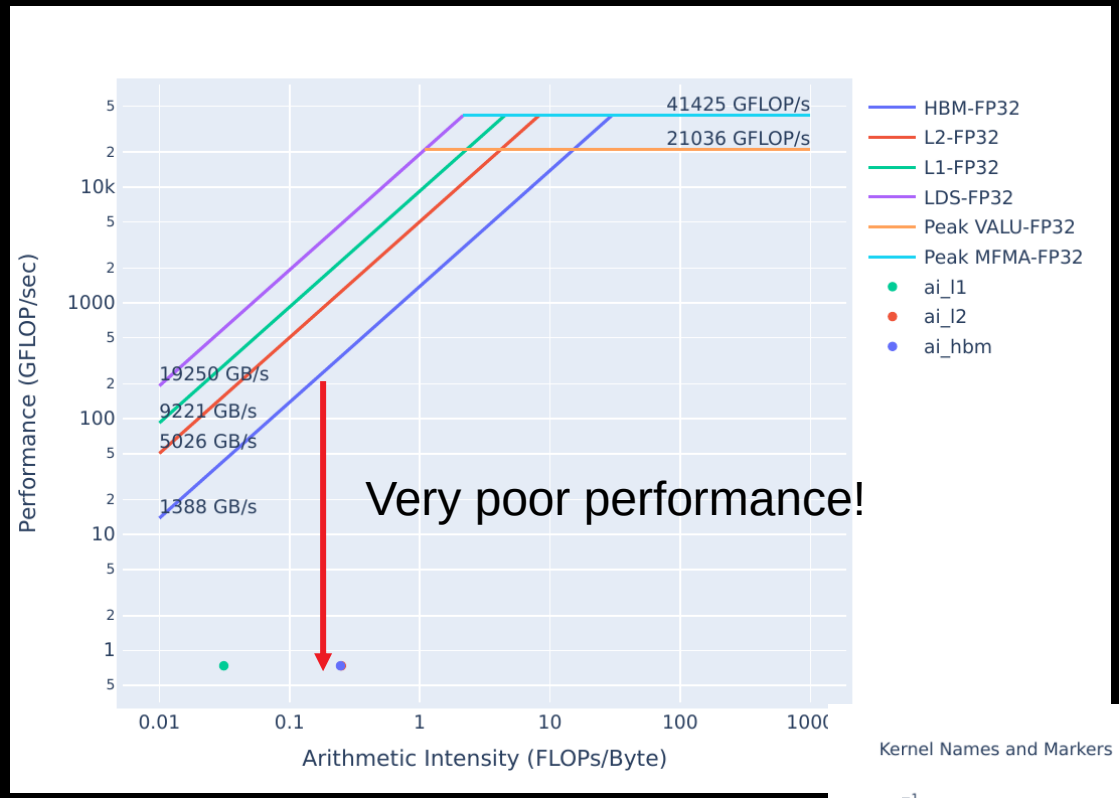
- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `omniperf profile –n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/mi200` folder.
    - `--roof-only` generates PDF roofline plots, and does **not** generate any non-roofline profiling data
    - `--kernel-names` generates a PDF showing which kernel names correspond to which icons in the roofline

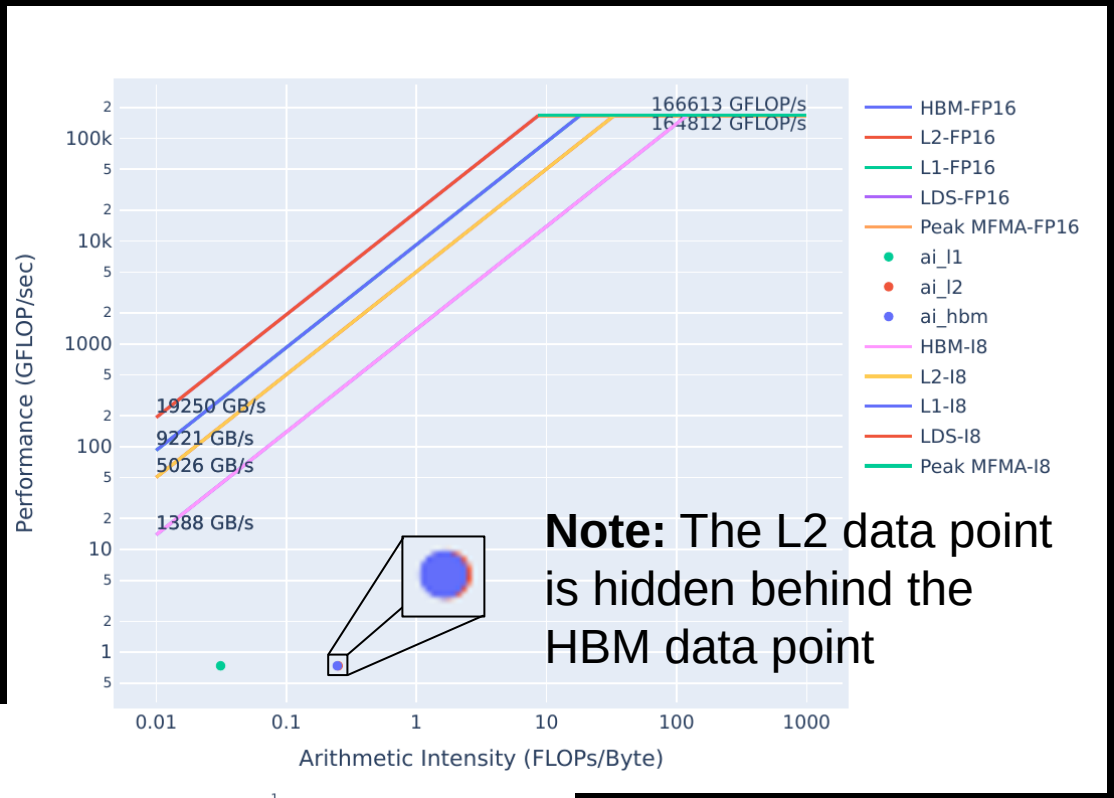
- Rooflines are a useful tool in determining which kernels are good optimization targets
  - They are only one perspective of performance: runtime of the kernel cannot be inferred from the roofline

- 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

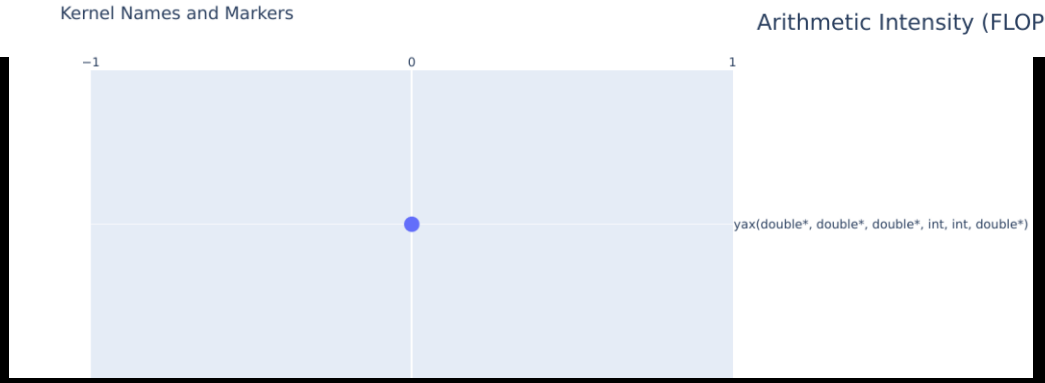
Exercise 1: Problem Roofline Plots





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

Kernel legend



Exercise 1: Prep to use Omniperf 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 the launch parameters for a given kernel can be obfuscated

- Omniperf can easily show launch parameter information regardless of the code
  - You just need the dispatch ID

- To generate profiling data, use the commands:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf 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 to profile** – use smaller representative problems if possible

Exercise 1: CLI Omniperf Comparisons are Easy

```
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 7.1.0 7.1.1 7.1.2
```

```
--------
Analyze
--------
```

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

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	754934306.50	69702016.5 (-90.77%)	754934306.50	69702016.5 (-90.77%)	754934306.50	69702016.5 (-90.77%)	100.00	100.0 (0.0%)

10.8x speedup

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

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
7.1.0	Grid Size	256.00	131072.0 (51100.0%)	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%)	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%)	4.00	2048.0 (51100.0%)	4.00	2048.0 (51100.0%)	Wavefronts

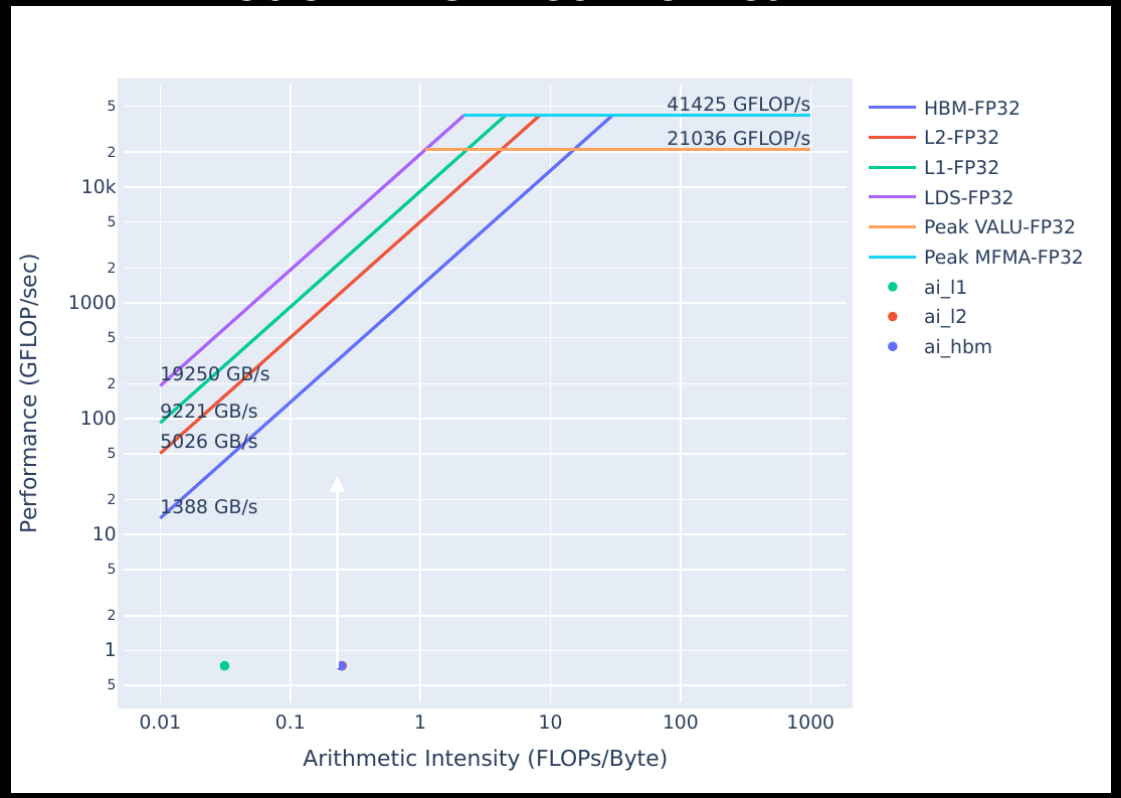
Increased launched wavefronts, which increases Grid Size

In general, it is difficult to pre-determine optimal launch bounds, so some experimentation is likely necessary

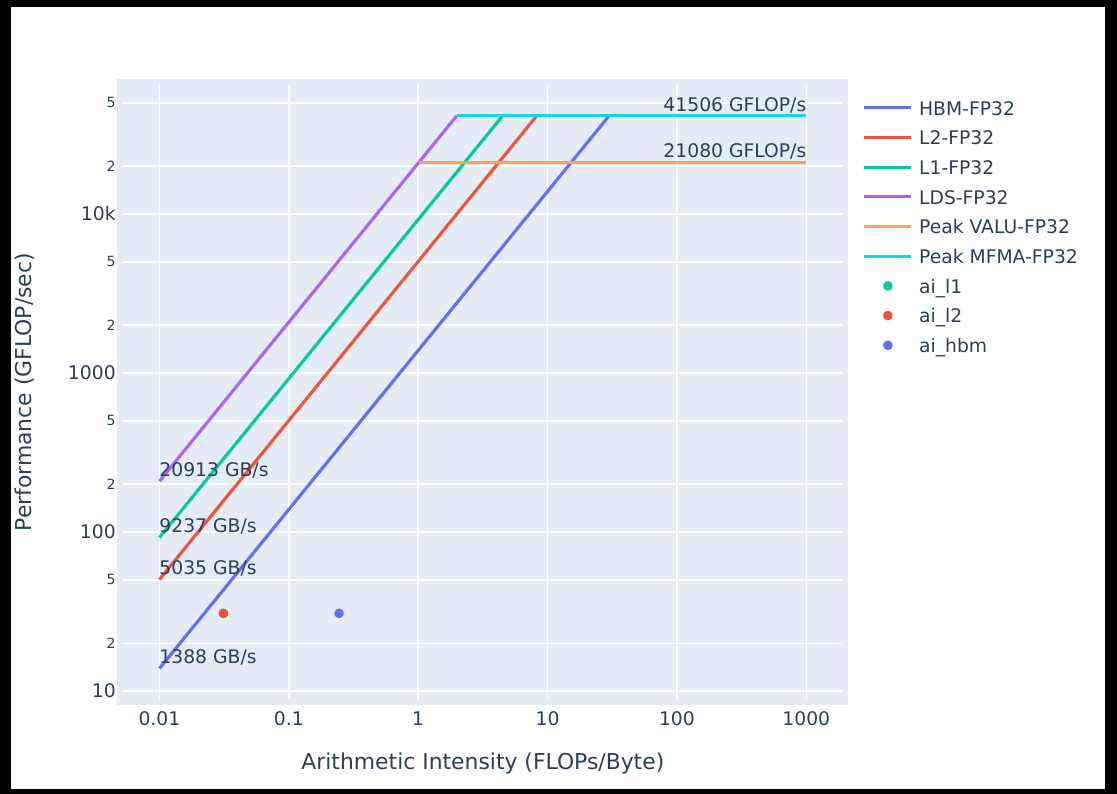
These slides always put `problem` as the baseline, and `solution` as the comparative

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 with Omniperf

- Use this omniperf command to check launch parameters:
  - `omniperf analyze –p workloads/problem/mi200 --dispatch 1 --metric 7.1.0 7.1.1 7.1.2`
    - Shows the launch parameters of the kernel with dispatch ID 1
    - `--metric` filters the output to **only** show these launch parameters

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

- It can be difficult to track down where launch parameters are set in code

- Omniperf can easily show the launch parameters of a kernel
  - Need the dispatch ID or index given by `--list-kernels`
  - `--list-kernels` index can be passed to `–k` as in:
    - `omniperf analyze –p workloads/problem/mi200 –k 0 –metric 7.1.0 7.1.1 7.1.2`

- **Note:**
  - These metric numbers are for Omniperf 1.0.10

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 Omniperf output in the next slide
- Omniperf makes it easy to determine if LDS allocations restrict occupancy, as before profile with:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 2: LDS Occupancy Limiter – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 2.1.26 6.2.7
```

```
--------
Analyze
--------
```

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

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	175427205.00	50366185.0 (-71.29%)	175427205.00	50366185.0 (-71.29%)	175427205.00	50366185.0 (-71.29%)	100.00	100.0 (0.0%)

3.4x speedup

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

Index	Metric	Value	Value	Unit	Peak	Peak	PoP	PoP
2.1.26	Wave Occupancy	102.70	487.32 (374.51%)	Wavefronts	3328.00	3328.0 (0.0%)	3.09	14.64 (373.88%)

+ ~11% Occupancy (overall)

```
----------------------------------------------------------------------------
6. Shader Processor Input (SPI)
6.2 SPI Resource Allocation
```

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
6.2.7	Insufficient CU LDS	6015745446.00	0.0 (-100.0%)	6015745446.00	0.0 (-100.0%)	6015745446.00	0.0 (-100.0%)	Cu

Sharp decrease in SPI stat

Exercise 2: Use SPI Stats to Determine if LDS Limits Occupancy

- Occupancy limiters can negatively impact performance

- Workgroup manager (SPI) stats in Omniperf indicate whether a kernel resource limits occupancy

- You can get the SPI stat for LDS for a single kernel with:
  - `omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 2.1.26 6.2.7`

Note:

- In current Omniperf release 1.0.10, the SPI “insufficient resource” stats are a count of cycles, meaning:
  - Large numbers (on the order of over 1 million) are expected if a field is not zero
  - The magnitude of these fields **does not** necessarily indicate how severely occupancy is impacted
  - If two fields are nonzero, the larger number indicates that resource is limiting occupancy more

- In a coming release, these “insufficient resource” fields are changing to percentages:
  - Large numbers will no longer be expected, but the other points will still hold

Exercise 3: Diagnosing a Register Occupancy Limiter

- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - In this case an `assert` on line 15 causes occupancy to be limited by register usage
  - The solution simply removes the assert

- 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:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 3: Register Occupancy Limiter – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 2.1.26 6.2.5 7.1.5 7.1.6 7.1.7
```

0. Top Stat

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	76983902.00	69815871.0 (-9.31%)	76983902.00	69815871.0 (-9.31%)	76983902.00	69815871.0 (-9.31%)	100.00	100.0 (0.0%)

Minor speedup

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

Index	Metric	Value	Value	Unit	Peak	Peak	PoP	PoP
2.1.26	Wave Occupancy	438.00	444.1 (1.39%)	Wavefronts	3328.00	3328.0 (0.0%)	13.16	13.34 (1.4%)

Small increase in occupancy

6. Shader Processor Input (SPI)
6.2 SPI Resource Allocation

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
6.2.5	Insufficient SIMD VGPRs	13733460.00	0.0 (-100.0%)	13733460.00	0.0 (-100.0%)	13733460.00	0.0 (-100.0%)	Simd

Large decrease in SPI stat

7. Wavefront
7.1 Wavefront Launch Stats

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
7.1.5	VGPRs	92.00	32.0 (-65.22%)	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	0.0 (-100.0%)	132.00	0.0 (-100.0%)	Registers
7.1.7	SGPRs	48.00	96.0 (100.0%)	48.00	96.0 (100.0%)	48.00	96.0 (100.0%)	Registers

Able to use:
Fewer VGPRs,
No AGPRS,
more SGPRs

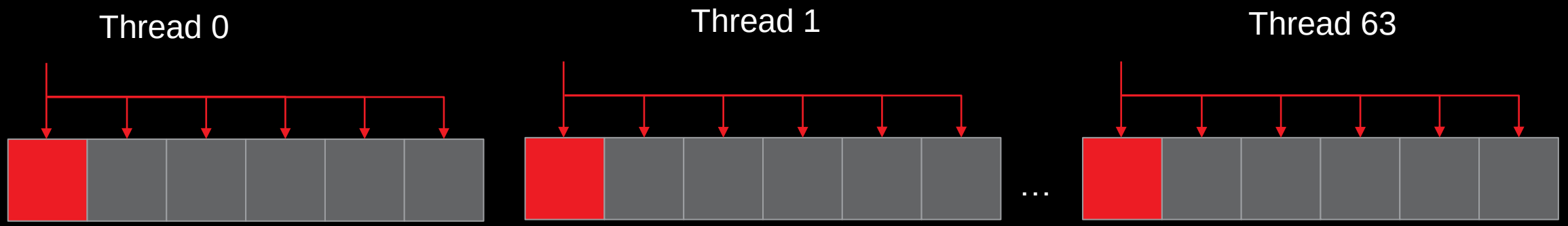
Exercise 3: Register Occupancy Limiter - Takeaways

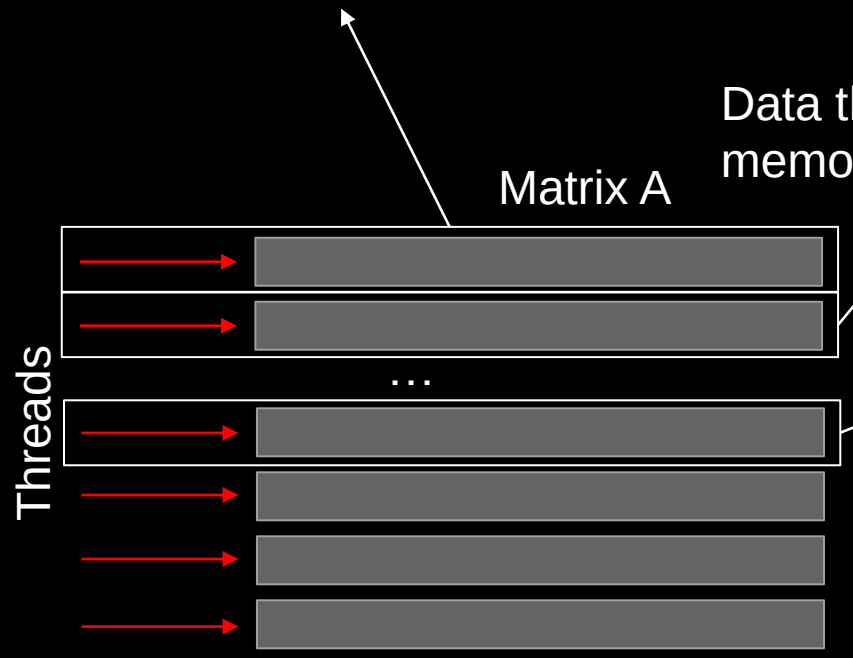
- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - Asserts, and even excessive use of math functions in kernels can degrade performance
- In this case the occupancy limit was very minor, despite a large number in the SPI stat
- AGPR usage in the absence of MFMA (Matrix Fused Multiply Add) instructions can indicate degraded performance.
  - Spilling registers to AGPRs, due to running out of VGPRs
- To determine if any SPI “insufficient resource” stats are nonzero, you can do:
  - `omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 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 Omniperf can quickly show if a kernel’s data access is adversarial to the caches

Exercise 4: What is a “Strided Data Access Pattern”?

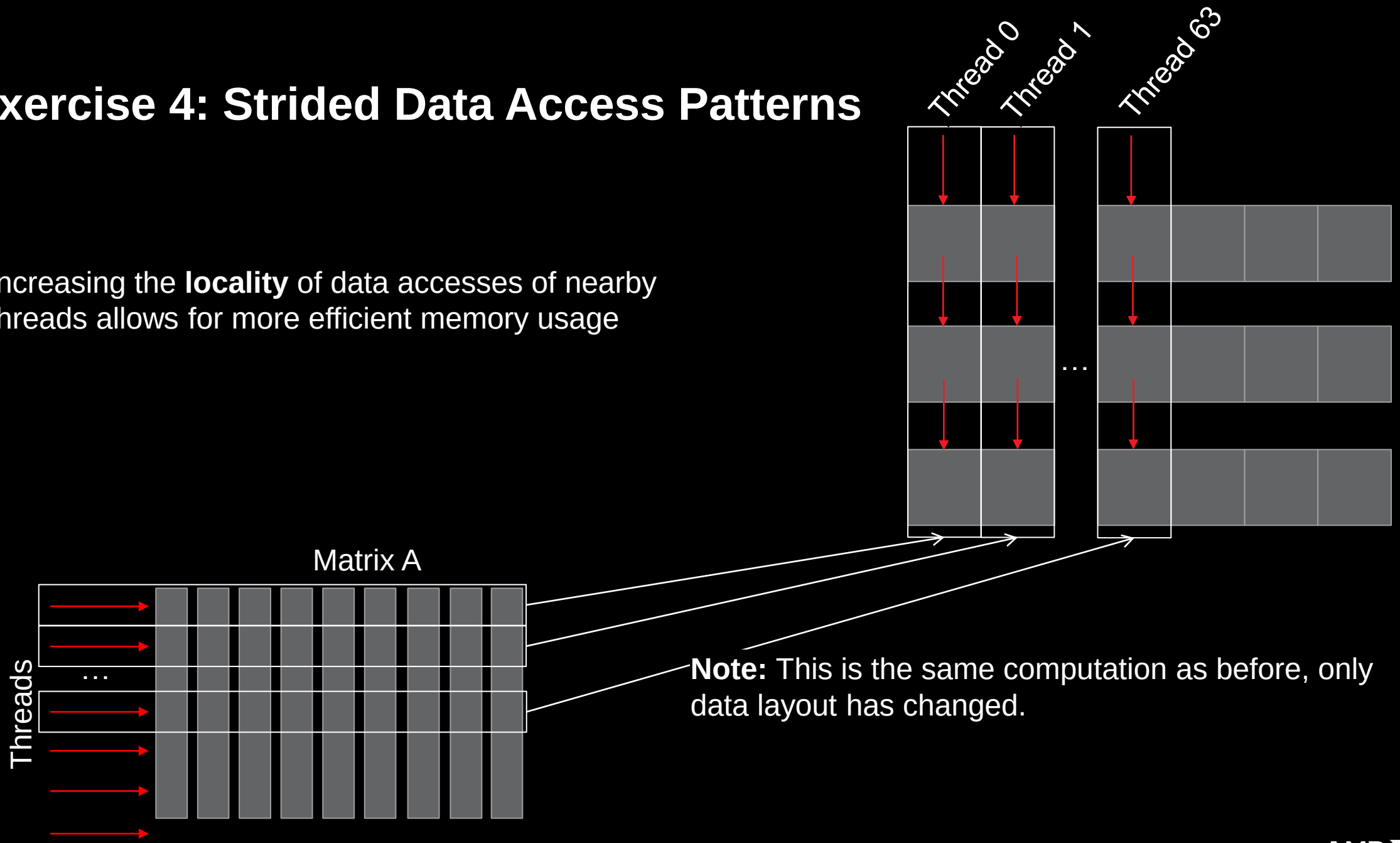




Data that each thread accesses at each step requires striding through memory, which leads to sub-optimal memory system usage.

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: Using Omniperf to 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 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

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

Exercise 4: Strided Data Access Pattern – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 16.1 17.1
```

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

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	69875592.00	12469690.5 (-82.15%)	69875592.00	12469690.5 (-82.15%)	69875592.00	12469690.5 (-82.15%)	100.00	100.0 (0.0%)

5.6x speedup

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

Index	Metric	Value	Value	Unit
16.1.0	Buffer Coalescing	25.00	25.0 (0.0%)	Pct of peak
16.1.1	Cache Util	87.80	98.08 (11.7%)	Pct of peak
16.1.2	Cache BW	8.69	12.18 (40.19%)	Pct of peak
16.1.3	Cache Hit	0.00	49.98 (inf%)	Pct of peak

+ ~50% in L1 hit

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

Index	Metric	Value	Value	Unit
17.1.0	L2 Util	98.74	98.39 (-0.36%)	Pct
17.1.1	Cache Hit	93.45	0.52 (-99.44%)	Pct
17.1.2	L2-EA Rd BW	125.69	688.98 (448.16%)	Gb/s
17.1.3	L2-EA Wr BW	0.00	0.0 (inf%)	Gb/s

L2 Cache Hit decreases sharply, Read BW from HBM increases by ~5x

The solution better uses the L1, but our L2 hit rate has degraded, which points to a deficiency in our algorithm

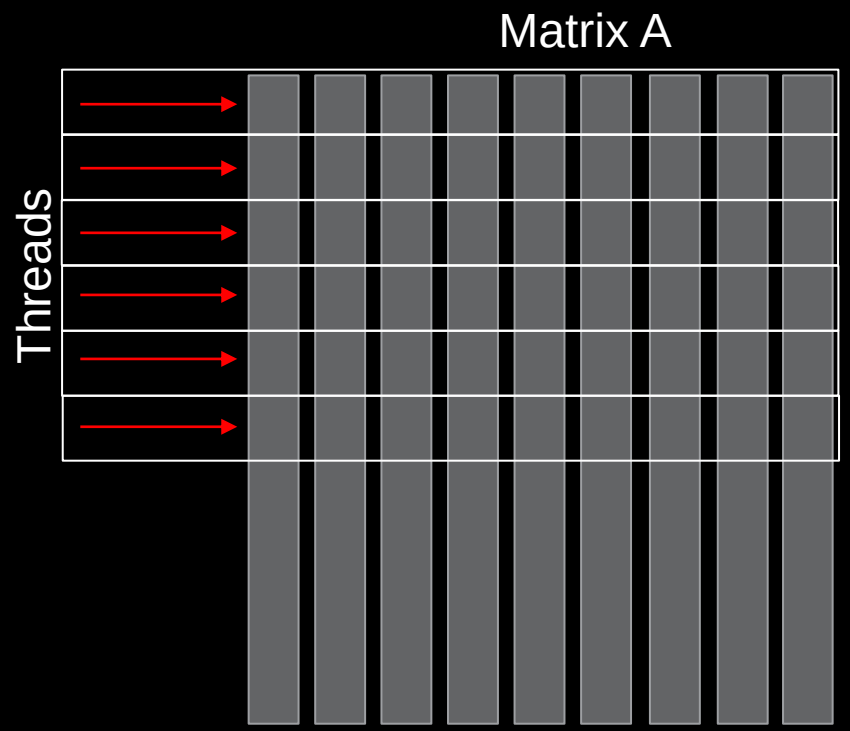
Exercise 4: Omniperf Speed-of-Light Cache Access Statistics

- This Omniperf command will show high-level details about L1 and L2 cache accesses:
  - `omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 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
- **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

Exercise 5: Algorithmic Optimizations

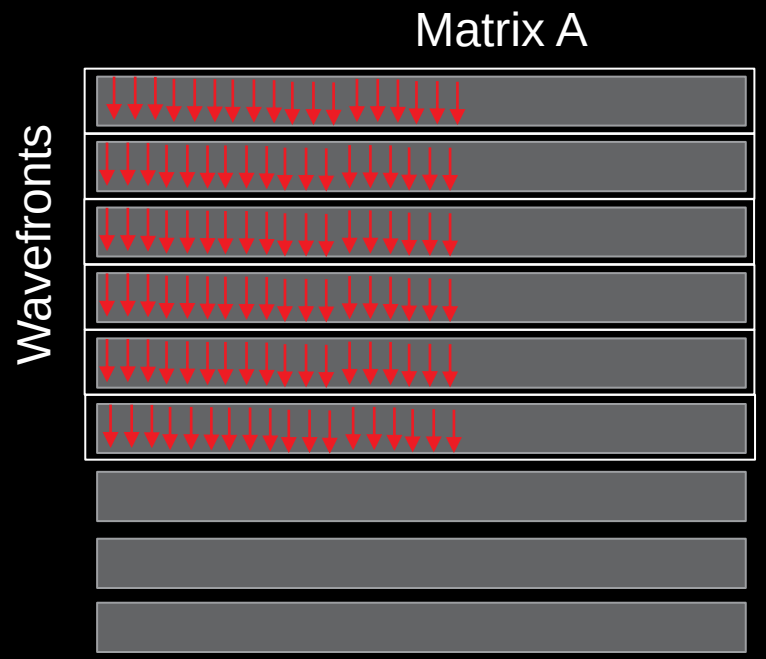
- These types of optimizations are the most difficult to execute
  - Generally, it is difficult to determine if the runtime of one algorithm will be faster than another
- We start with the solution from last exercise as our problem
  - Speed-of-light cache statistics showed that we had ~0% hit rate in the L2, could it be better?
- Our initial algorithm is naïve in terms of parallelization:
  - Each thread computes the sum of a row
- Exposing more parallelism is possible and should get us more performance in this case

Exercise 5: Algorithmic Optimizations



In our current algorithm, each thread computes the sum of a single row

Exercise 5: Algorithmic Optimizations



In a more efficient implementation, wavefronts have multiple threads sum up the rows in parallel, using shared memory to reduce partial sums

Note: The original data layout allows the wavefronts to avoid striding memory

Exercise 5: Using Omniperf to Evaluate an Algorithmic Optimization

- The strided data access pattern issue is everywhere
  - This solution gets about 2x faster when the data layout is switched to optimize locality

- Though the solution shows a **29x speedup** from the problem, cache speed-of-light stats aren't convincing
  - The rooflines for these problems do not tell the full performance story either

- Running the solution shows it is much faster, but does it use the caches more efficiently?

- To get started, run:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 5: Sometimes the Full Story is in the Details

```
omniperf analyze -p workloads/problem/mi200 -p workloads/solution/mi200 --dispatch 1 --metric 16.3 17.2 17.3
```

0. Top Stat

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	12443928.00	408316.0 (-96.72%)	12443928.00	408316.0 (-96.72%)	12443928.00	408316.0 (-96.72%)	100.00	100.0 (0.0%)

~29x faster

16. Vector L1 Data Cache
16.3 L1D Cache Accesses

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
16.3.0	Total Req	524368.00	16448.0 (-96.86%)	524368.00	16448.0 (-96.86%)	524368.00	16448.0 (-96.86%)	Req per wave	- ~32x
...									
16.3.5	Cache Accesses	131140.00	4097.0 (-96.88%)	131140.00	4097.0 (-96.88%)	131140.00	4097.0 (-96.88%)	Req per wave	- ~32x
16.3.6	Cache Hits	65538.00	2864.0 (-95.63%)	65538.00	2864.0 (-95.63%)	65538.00	2864.0 (-95.63%)	Req per wave	
16.3.7	Cache Hit Rate	49.98	69.9 (39.87%)	49.98	69.9 (39.87%)	49.98	69.9 (39.87%)	Pct	**+ ~40%**
...									

17. L2 Cache
17.2 L2 - Fabric Transactions

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
17.2.0	Read BW	4194916.56	65688.69 (-98.43%)	4194916.56	65688.69 (-98.43%)	4194916.56	65688.69 (-98.43%)	Bytes per wave	- ~64x
...									

17.3 L2 Cache Accesses

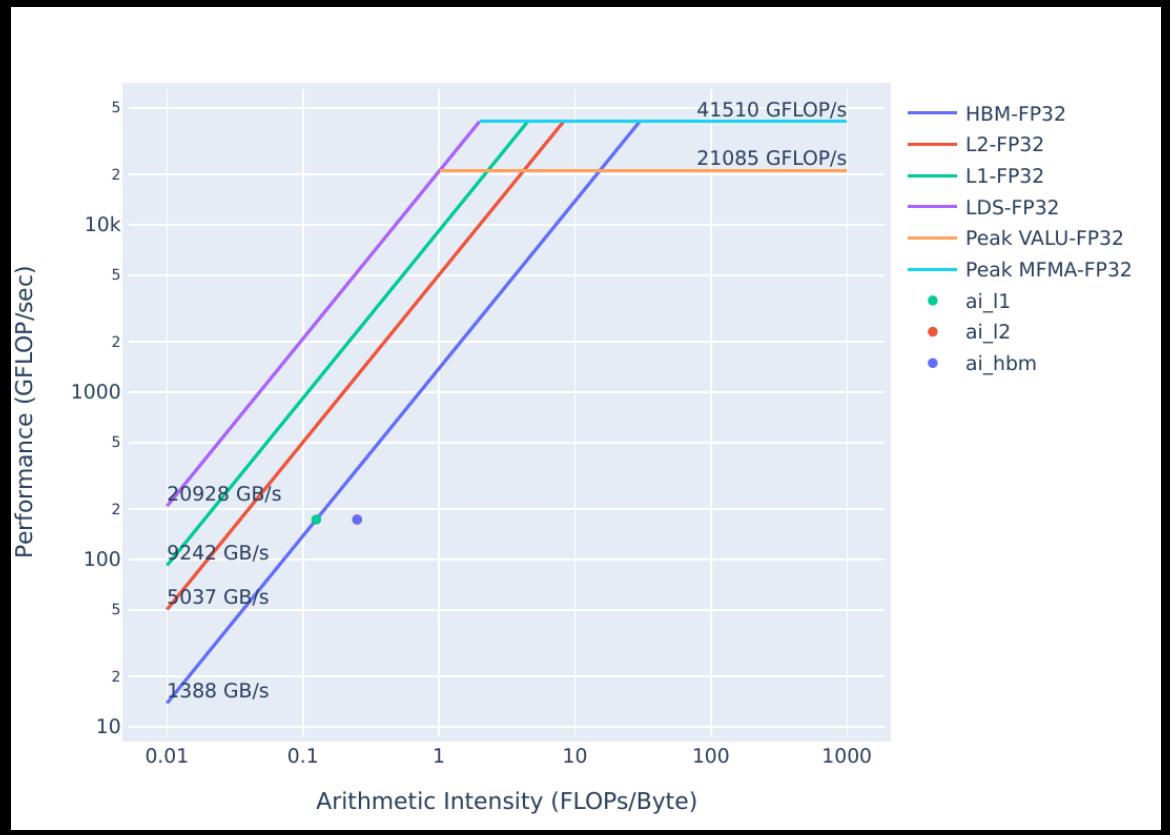
Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
17.3.0	Req	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	Req per wave	- ~53x
...									
17.3.6	Hits	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	Hits per wave	
17.3.7	Misses	32774.06	513.38 (-98.43%)	32774.06	513.38 (-98.43%)	32774.06	513.38 (-98.43%)	Misses per wave	- ~64x
17.3.8	Cache Hit	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	Pct	**Large relative gain, + ~16% overall**
...									

Cache hit rates alone do not give a convincing reason for our performance increase

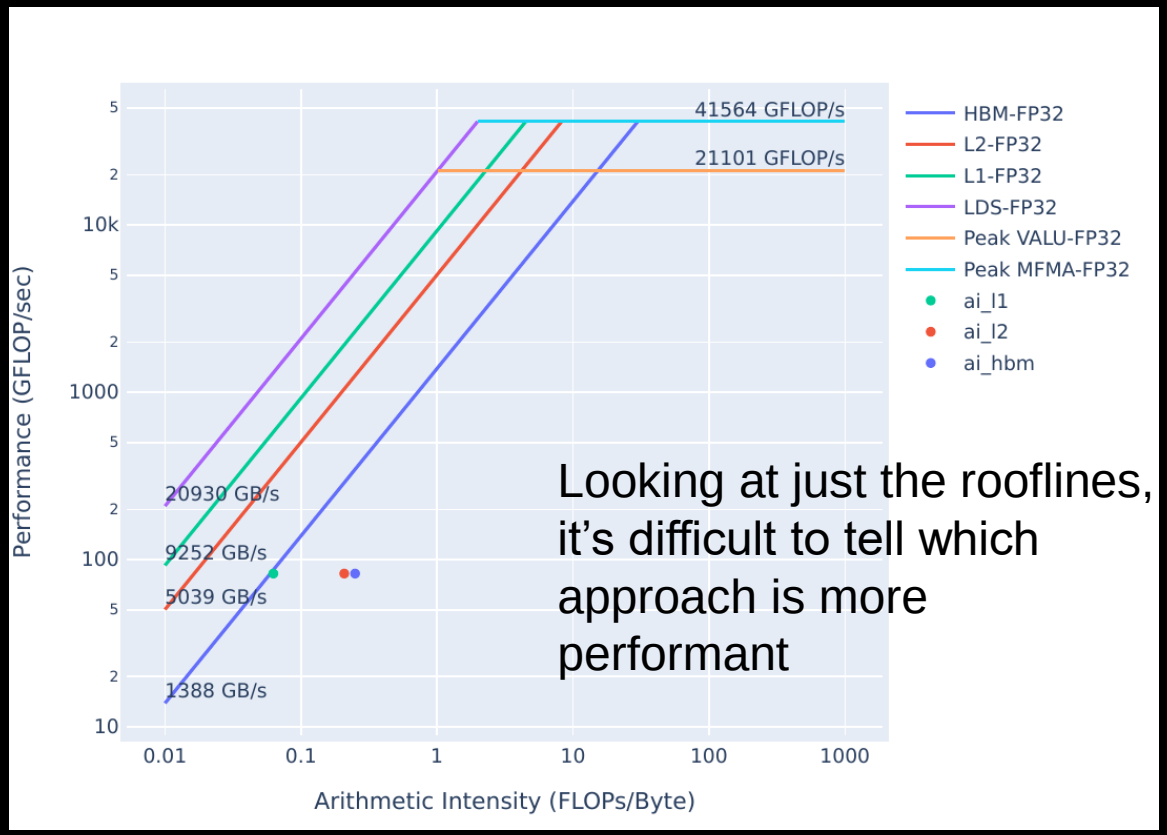
Exercise 5: It Can Be Hard to Compare Rooflines Between Algorithms

- `omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
- `omniperf profile -n solution_roof_only --roof-only --kernel-names -- ./solution.exe`

Problem FP32 Roofline



Solution FP32 Roofline



Looking at just the rooflines, it's difficult to tell which approach is more performant

`problem` is closer to being HBM bandwidth bound: It needs to request much more data from HBM than the optimized version

Exercise 5: Omniperf Detailed Cache Statistics - Takeaways

- To get detailed cache statistics (including data movement) for kernel with dispatch ID 1:
  - `omniperf analyze -p workloads/problem/mi200 --dispatch 1 --metric 16.2 16.3 17.2 17.3`
    - **Note:** The slide omitted some Omniperf output from this metric filtering

- Algorithmic optimizations can be powerful, but are usually time-intensive to design and implement

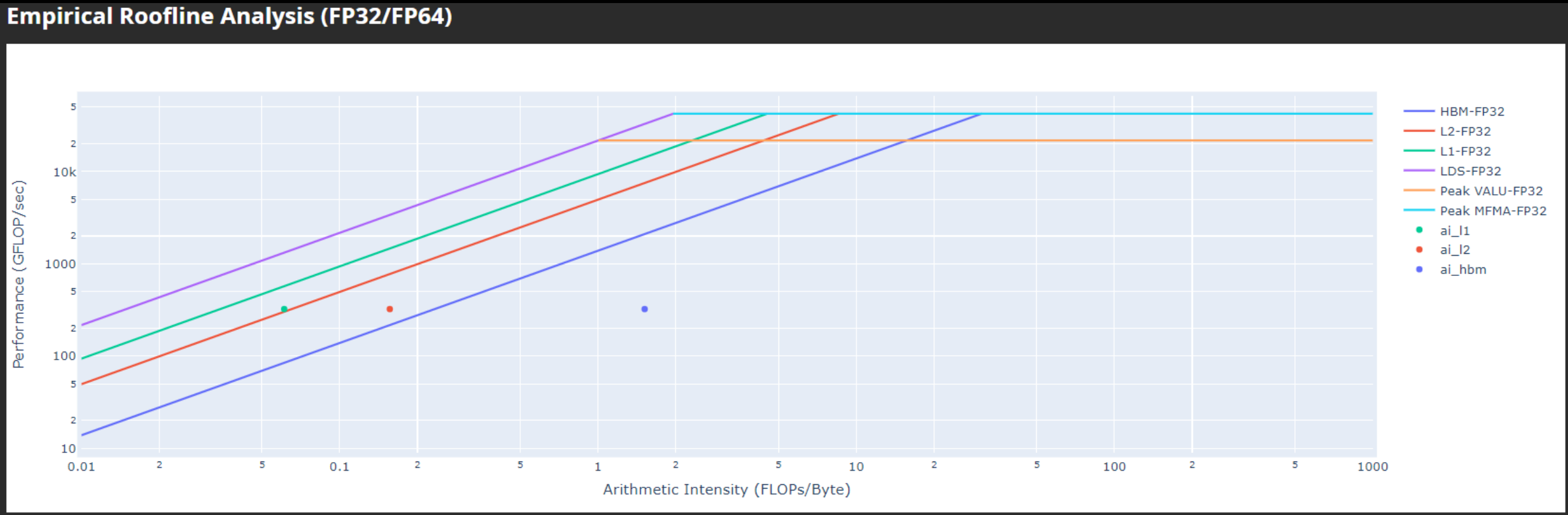
- It can be difficult to understand the performance differences between algorithms
  - Rooflines can be misleading
  - Assuming correctness is verified, timings don’t lie
  - Detailed profiling data can help shed light on the *why* of performance differences

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



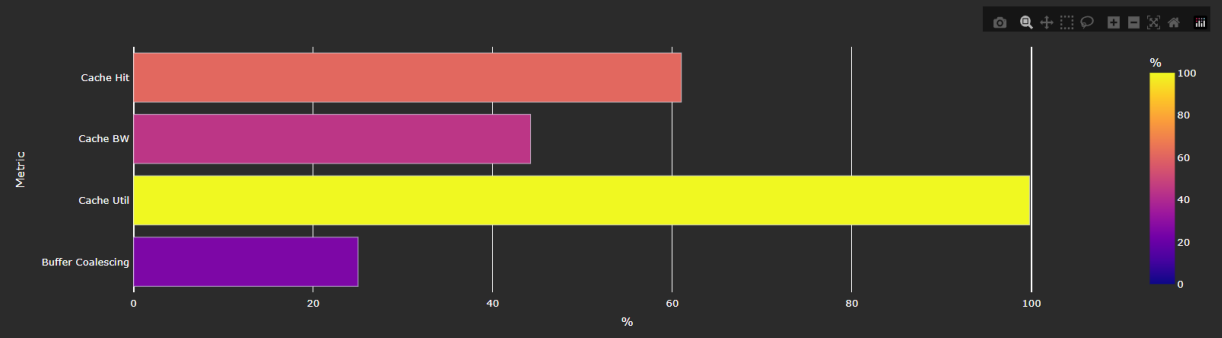
- Performance: almost 330 GFLOPs

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

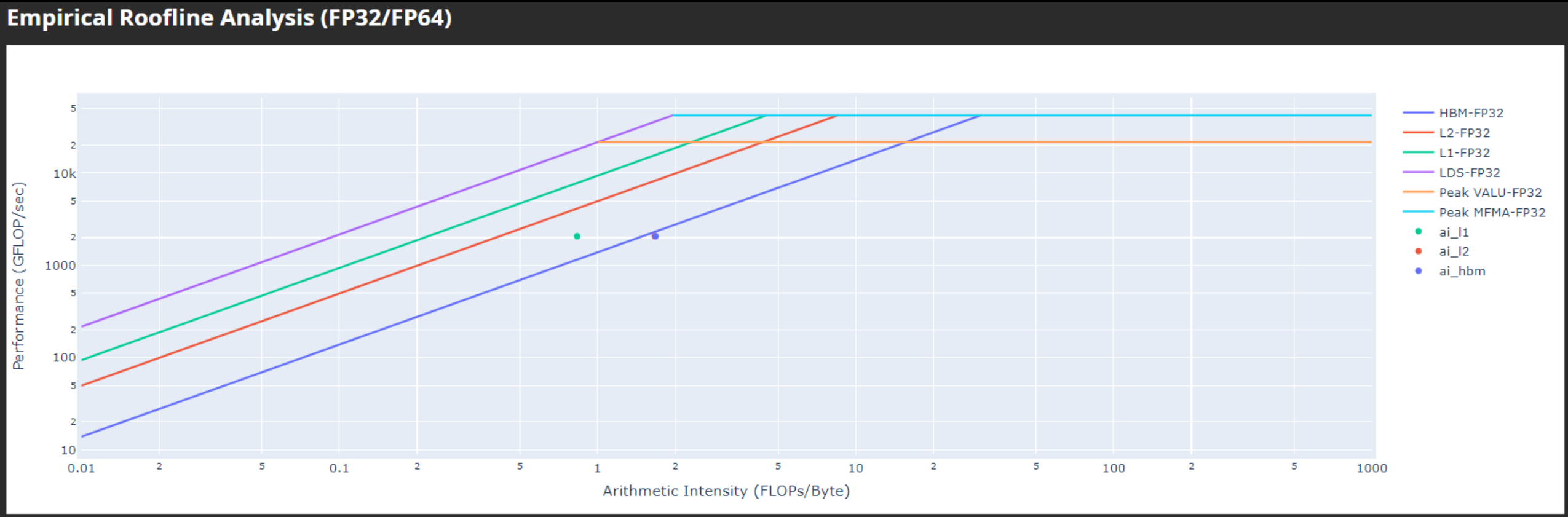
16.3 L1D Cache Accesses

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



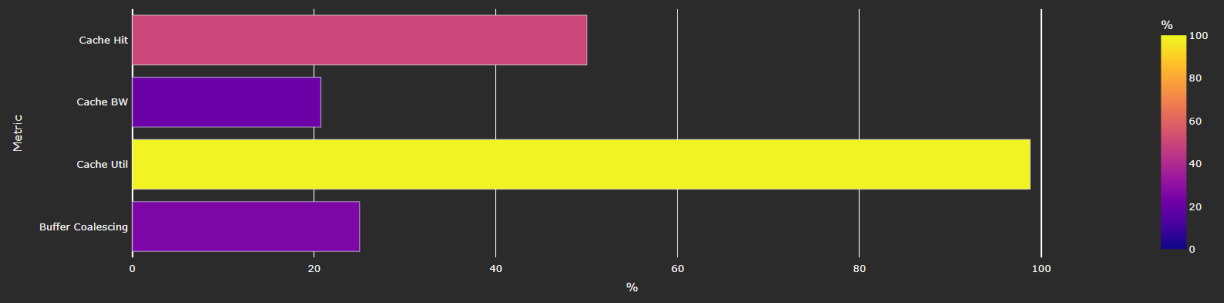
- Performance: almost 2 TFLOPs

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

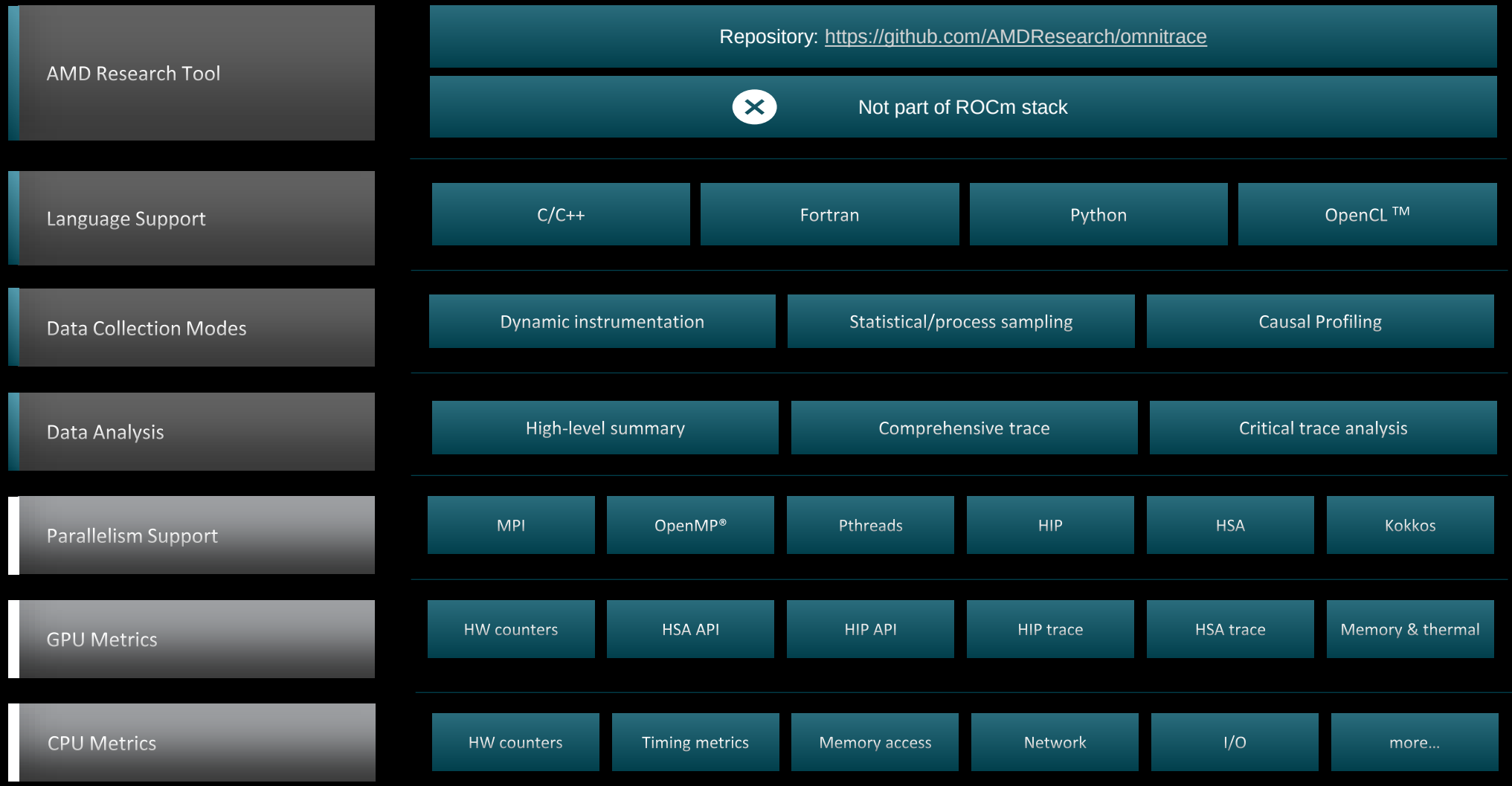
Metric	Mean	Min	Max	unit
Stalled on L2 Data	79.08	79.08	79.08	Pct
Stalled on L2 Req	15.17	15.17	15.17	Pct
Tag RAM Stall (Read)	0.00	0.00	0.00	Pct
Tag RAM Stall (Write)	0.00	0.00	0.00	Pct
Tag RAM Stall (Atomic)	0.00	0.00	0.00	Pct

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

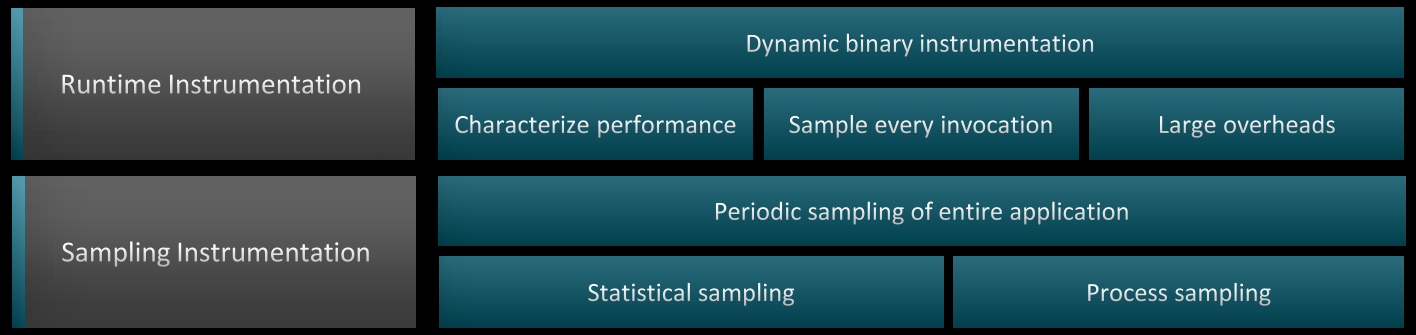
Omnitrace

Omnitrace: Application Profiling, Tracing, and Analysis



Refer to current documentation for recent updates

Omnitrace instrumentation Modes



Basic command-line syntax:

```
$ omnitrace [omnitrace-options] -- <CMD> <ARGS>
```

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

```
$ omnitrace -h
```

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

```
$ srun [options] omnitrace [omnitrace-options] -- <CMD> <ARGS>
```

For problems, create an issue here: https://github.com/AMDResearch/omnitrace/issues
Documentation: https://amdresearch.github.io/omnitrace/

Omnitrace Configuration

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

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

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

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

```
$ omnitrace-avail –c perfetto
```

```
$ omnitrace-avail -c perfetto
|-----------------------------------|-----------------|-------------------------------------------------------------------|
|        ENVIRONMENT VARIABLE       |      VALUE      |                            CATEGORIES                             |
|-----------------------------------|-----------------|-------------------------------------------------------------------|
| OMNITRACE_PERFETTO_BACKEND        |    inprocess    | custom, libomnitrace, omnitrace, perfetto                         |
| OMNITRACE_PERFETTO_BUFFER_SIZE_KB |     1024000     | custom, data, libomnitrace, omnitrace, perfetto                   |
| OMNITRACE_PERFETTO_FILL_POLICY    |     discard     | custom, data, libomnitrace, omnitrace, perfetto                   |
| OMNITRACE_TRACE_DELAY             |        0        | custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace |
| OMNITRACE_TRACE_DURATION          |        0        | custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace |
| OMNITRACE_TRACE_PERIODS           |                 | custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace |
| OMNITRACE_TRACE_PERIOD_CLOCK_ID   | CLOCK_REALTIME  | custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace |
| OMNITRACE_USE_PERFETTO            |      true       | backend, custom, libomnitrace, omnitrace, perfetto                |
|-----------------------------------|-----------------|-------------------------------------------------------------------|
```

Shows all runtime settings that may be tuned for perfetto

Omnitrace Configuration

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

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

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

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

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

For brief description, use the options:

```
$ omnitrace-avail –bd
```

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

Create a config file

Create a config file in $HOME:

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

To add description of all variables and settings, use:

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

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

```
<snip>
OMNITRACE_USE_PERFETTO                          = true
OMNITRACE_USE_TIMEMORY                          = true
OMNITRACE_USE_SAMPLING                          = false
OMNITRACE_USE_ROCTRACER                         = true
OMNITRACE_USE_ROCM_SMI                          = true
OMNITRACE_USE_KOKKOSP                           = false
OMNITRACE_USE_CAUSAL                            = false
OMNITRACE_USE_MPIP                              = true
OMNITRACE_USE_PID                               = true
OMNITRACE_USE_ROCPROFILER                       = true
OMNITRACE_USE_ROCTX
<snip>
```

Contents of the config file

Declare which config file to use by setting the environment:

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

Dynamic Instrumentation

Binary Rewrite

Binary Rewrite – Jacobi Example

Binary Rewrite

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

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

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

This new binary will have instrumented functions

Subroutine Instrumentation

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

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

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

Path to new instrumented binary

Binary Rewrite – Jacobi Example

Binary Rewrite

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

Generating a new /library with instrumentation built-in:

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

Run the instrumented binary:

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

subroutine instrumentation

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

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

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

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

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

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

Generates traces for application run

List of Instrumented GPU Functions

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

ROCM TRACER (ACTIVITY API)

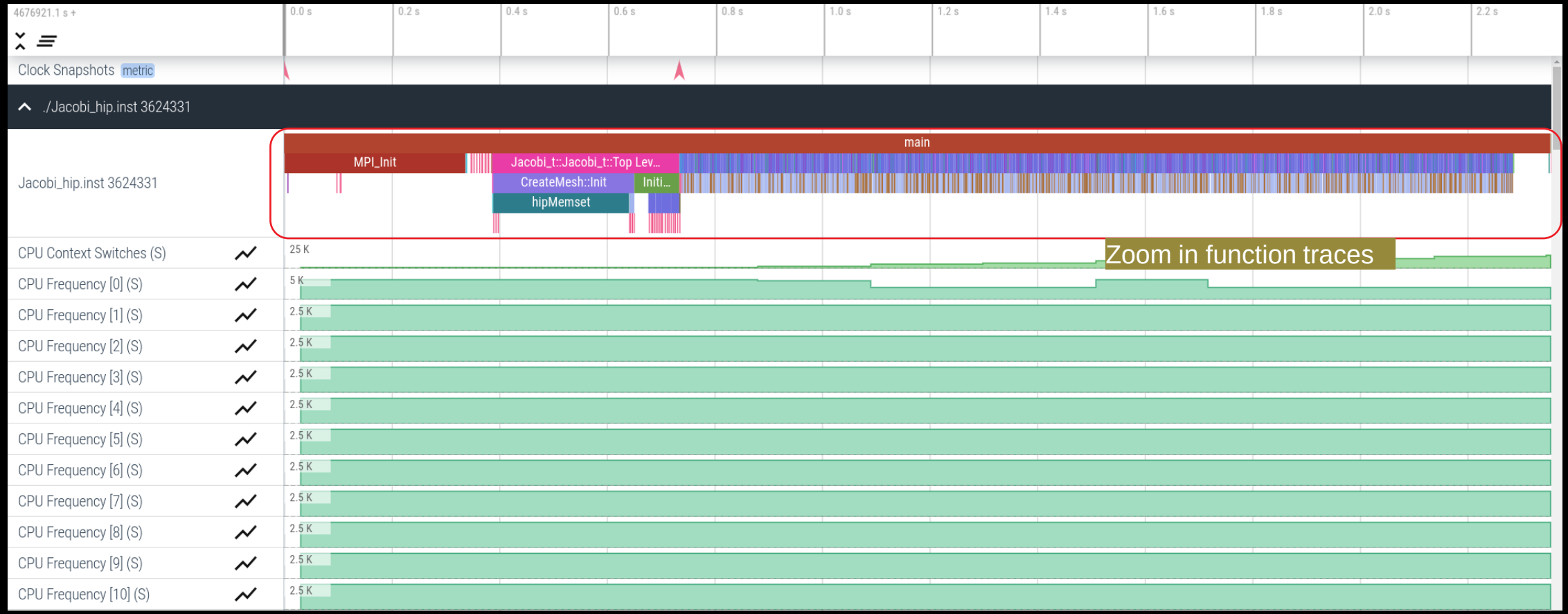
LABEL	COUNT	DEPTH	METRIC	UNITS	SUM	MEAN	% SELF
|0>>> pthread_create	1	0	roctracer	sec	0.000353	0.000353	0.0
|1>>> |_start_thread	1	1	roctracer	sec	2.344864	2.344864	100.0
|0>>> hipInit	1	0	roctracer	sec	0.000000	0.000000	0.0
|0>>> hipGetDeviceCount	1	0	roctracer	sec	0.000000	0.000000	0.0
|0>>> hipSetDevice	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



Hardware Counters

Hardware Counters – List All

```
$ mpirun -np 1 omnitrace-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
OMNITRACE_CAUSAL_BINARY_EXCLUDE		string	Excludes binaries matching the list of pr...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_BINARY_SCOPE	%MAIN%	string	Limits causal experiments to the binaries...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_DELAY	0	double	Length of time to wait (in seconds) befor...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_DURATION	0	double	Length of time to perform causal experime...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_FUNCTION_EXCLUDE		string	Excludes functions matching the list of p...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_FUNCTION_SCOPE		string	List of <function> regex entries for caus...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_RANDOM_SEED	0	unsigned long	Seed for random number generator which se...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_SOURCE_EXCLUDE		string	Excludes source files or source file + li...	analysis, causal, custom, libomnitrace, o...
OMNITRACE_CAUSAL_SOURCE_SCOPE		string	Limits causal experiments to the source f...	analysis, causal, custom, libomnitrace, o...

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		Level 1 cache misses

CPU Hardware Counters

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

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	true	Total wavefronts.
VALUInsts:device=0	true	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...

GPU Hardware Counters

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

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

Modify config file

Create a config file in $HOME:

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

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

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

To profile desired metrics for all participating GPUs:

```
…
OMNITRACE_ROCM_EVENTS = GPUBusy, Wavefronts,
MemUnitBusy
…
```

Execution with Hardware Counters

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

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

```
[omnitrace][501266][0][omnitrace_finalize] Finalizing perfetto...
[omnitrace][501266][perfetto]> Outputting '/shared/prod/home/ssitaram/HPCTrainingExamples/HIP/jacobi/omnitrace-Jacobi_hip-output/2023-03-15_22.57/perfetto-trace-0.proto' (11
.. Done
[omnitrace][501266][rocprof-device-0-GPUBusy]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-GPUBusy-0.json'
[omnitrace][501266][rocprof-device-0-GPUBusy]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-GPUBusy-0.txt'
[omnitrace][501266][rocprof-device-0-Wavefronts]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-Wavefronts-0.json'
[omnitrace][501266][rocprof-device-0-Wavefronts]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-Wavefronts-0.txt'
[omnitrace][501266][rocprof-device-0-MemUnitBusy]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-MemUnitBusy-0.json'
[omnitrace][501266][rocprof-device-0-MemUnitBusy]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/rocprof-device-0-MemUnitBusy-0.txt'
[omnitrace][501266][trip_count]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/trip_count-0.json'
[omnitrace][501266][trip_count]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/trip_count-0.txt'
[omnitrace][501266][wall_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/wall_clock-0.json'
[omnitrace][501266][wall_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/wall_clock-0.txt'
[omnitrace][501266][roctracer]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/roctracer-0.json'
[omnitrace][501266][roctracer]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/roctracer-0.txt'
[omnitrace][501266][sampling_percent]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_percent-0.json'
[omnitrace][501266][sampling_percent]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_percent-0.txt'
[omnitrace][501266][sampling_cpu_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_cpu_clock-0.json'
[omnitrace][501266][sampling_cpu_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_cpu_clock-0.txt'
[omnitrace][501266][sampling_wall_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_wall_clock-0.json'
[omnitrace][501266][sampling_wall_clock]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_wall_clock-0.txt'
[omnitrace][501266][sampling_gpu_memory_usage]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_memory_usage-0.json'
[omnitrace][501266][sampling_gpu_memory_usage]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_memory_usage-0.txt'
[omnitrace][501266][sampling_gpu_power]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_power-0.json'
[omnitrace][501266][sampling_gpu_power]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_power-0.txt'
[omnitrace][501266][sampling_gpu_temperature]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_temperature-0.json'
[omnitrace][501266][sampling_gpu_temperature]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_temperature-0.txt'
[omnitrace][501266][sampling_gpu_busy_percent]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_busy_percent-0.json'
[omnitrace][501266][sampling_gpu_busy_percent]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/sampling_gpu_busy_percent-0.txt'
[omnitrace][501266][metadata]> Outputting 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/metadata-0.json' and 'omnitrace-Jacobi_hip-output/2023-03-15_22.57/functions-0.json'
[omnitrace][501266][0][omnitrace_finalize] Finalized: 31.657272 sec wall_clock,    0.000 MB peak_rss,  179.700 MB page_rss, 29.950000 sec cpu_clock,   94.6 % cpu_util
[889.832]       perfetto.cc:60129 Tracing session 1 ended, total sessions:0
```

GPU hardware counters

Tracing Multiple Ranks

Profiling Multiple MPI Ranks – Jacobi Example

Binary Rewrite

Generating a new /library with instrumentation built-in:

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

Run the instrumented binary with 2 ranks:

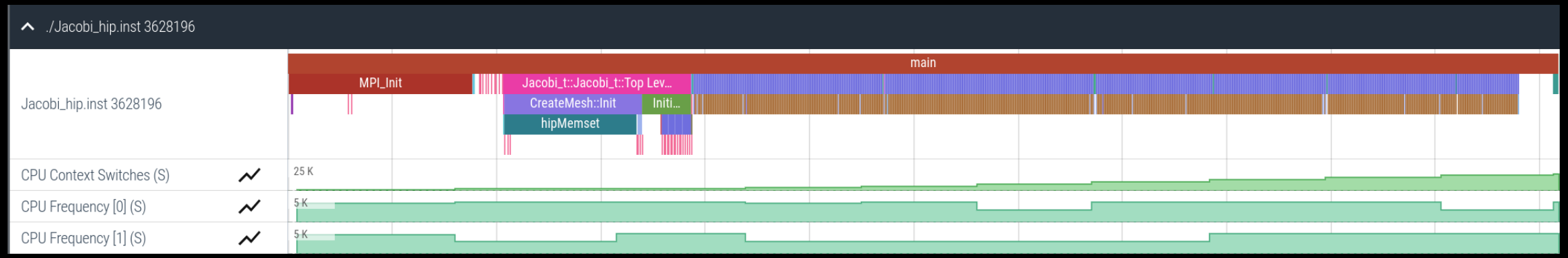
```
 $ mpirun -np 2 omnitrace-run --./Jacobi_hip.inst -g
2 1
```

```
[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'
```

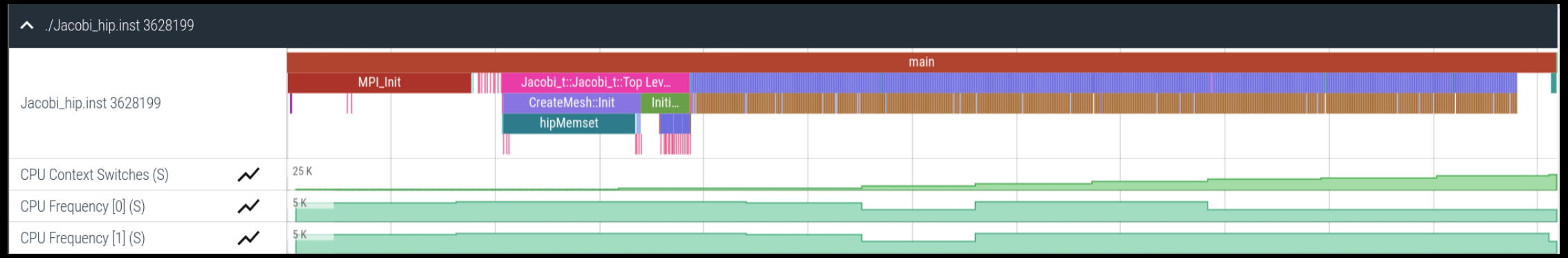
All output files are generated for each rank

Visualizing Traces from Multiple Ranks - Separately

MPI 0



MPI 1



Tips & Tricks

- My Perfetto timeline seems weird how can I check the clock skew?
  - Set `OMNITRACE_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 OMNITRACE_USE_ROCM_SMI=OFF
- What is the best way to profile multi-process runs?
  - Use OmniTrace'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=libomnitrace.so in the env. and set OMNITRACE_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, with v1.7.3 and later declare OMNITRACE_PERFETTO_ANNOTATIONS to false.

Summary

- OmniTrace 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

Questions?

DISCLAIMERS AND ATTRIBUTIONS


AMD