



Performance Profiling with Omniperf

Cole Ramos and Ian Bogle

HIP Training Series
OLCF
16 Oct 2023

AMD 
together we advance_



Cole Ramos



Cole.Ramos@amd.com

Software Development Engineer out of AMD's Research and Advanced Development group.

Omniperf project lead with prior experience building software at Microsoft and startups.

Ian Bogle



Ian.Bogle@amd.com

MTS Software Development Engineer – AMD's HPC Software Solutions group

- LAMMPS
- Kokkos

PhD in Graph Algorithms in HPC Contexts at Rensselaer Polytechnic Institute

- Algorithmic Design & Analysis
- Distributed Computing
- Kokkos

Agenda

General Overview

1. Introduction
2. Methodology
3. Implementation
 - Profile Mode
 - Analyze Mode
4. Roofline Analysis
5. Basic Examples

Guided Exercises

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

Reduce A+A Software Cost & Complexity

Reduce Cognitive Load

Reduce Time to *actionable* insights

Omnitrace

A tool for the entire execution of an application

- Generated high-level comprehensive trace of your application
- Identified particular kernels through a critical trace

Omniperf

A kernel-level performance analysis tool

- Extensive insight into the execution of individual kernels
- Examine kernel performance organized by logical IP blocks

Reduce A+A Software Cost & Complexity

Reduce Cognitive Load

Reduce Time to *actionable* insights

Omnitrace

A tool for the entire execution of an application

- Generated high-level comprehensive trace of your application
- Identified particular kernels through a critical trace

Omniperf

A kernel-level performance analysis tool

- Extensive insight into the execution of individual kernels
- Examine kernel performance organized by logical IP blocks

Omniperf

Repo + Docs

<https://github.com/AMDRResearch/omniperf>

Dependencies

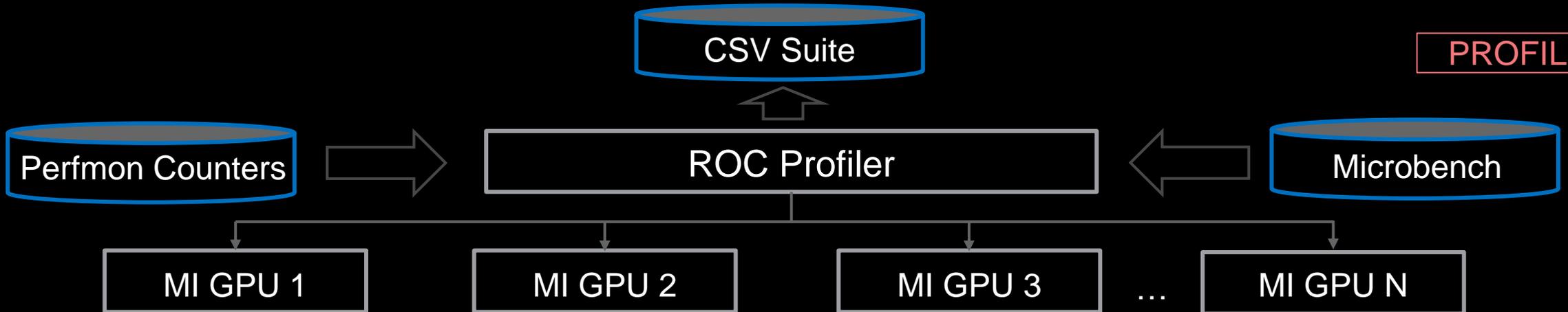
- ROCm (≥ 5.2)
- Python (≥ 3.7)
- CMake (≥ 3.19)

Support

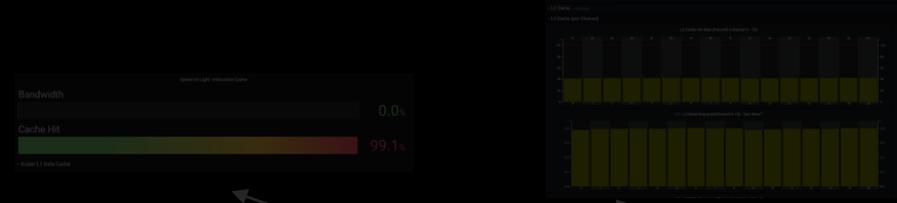
- All major Linux distributions
 - Ubuntu, SLES, RHEL
- Mi100, Mi200, Mi300 (in testing)

ANALYSIS

PROFILING



ANALYSIS



```
colanoso@pdp-1 ~$mpjert
```

```
colanoso@pdp-1 ~$python src/analyze.py analyze --workload/rocbench_all/rocbench_3_3
```

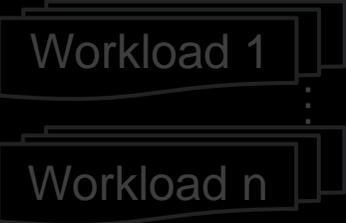
```
rocbench_3_3
```

```
Analysis
```

Rank	Name/Name	Count(Cnt)	Size(Cnt)	Peak(Cnt)	Median(Cnt)	Pct
0	__amd_vector_fill32fvec16float_id	1	233368	233368	233368	1
1	void benchmark_fvec16vec16_vector_typeFl...	1	180000	180000	180000	0
2	void benchmark_fvec16vec16_vector_typeFl...	1	179040	179040	179040	0
3	void benchmark_fvec16vec16_vector_typeFl...	1	179200	179200	179200	0
4	void benchmark_fvec16vec16_vector_typeFl...	1	402000	402000	402000	1
5	void benchmark_fvec16vec16_vector_typeFl...	1	179360	179360	179360	0
6	void benchmark_fvec16vec16_vector_typeFl...	1	179680	179680	179680	0
7	void benchmark_fvec16vec16_vector_typeFl...	1	179680	179680	179680	0
8	void benchmark_fvec16vec16_vector_typeFl...	1	180000	180000	180000	0
9	void benchmark_fvec16vec16_vector_typeFl...	1	180000	180000	180000	0

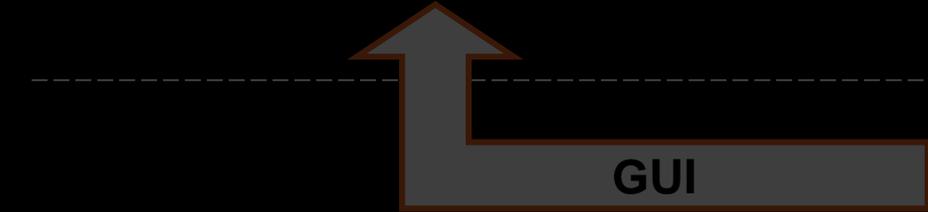


GUI Visualizer (Grafana)



DB Importer

CLI Analyzer



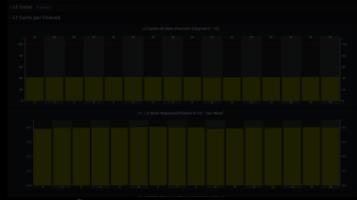
PROFILING



...



SERVER-SIDE



GUI Visualizer (Grafana)



Workload 1

Workload n

DB Importer

CLIENT-SIDE

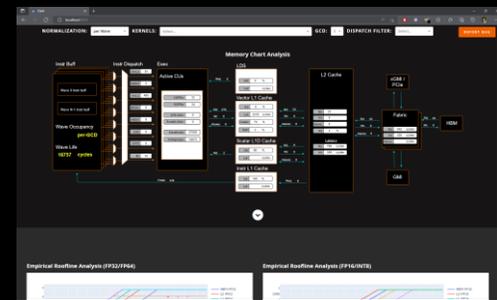
```
colanoso@pdp-1 ~$mpjart
```

```
colanoso@pdp-1:~/colanoso$ python3 src/analyze.py analyze --workload/colanoso_all/colanoso --s
```

```
colanoso@pdp-1:~/colanoso$ python3 src/analyze.py analyze --workload/colanoso_all/colanoso --s
```

Rank	Name/Name	Count(Cnt)	Size(Cnt)	Peak(Cnt)	Median(Cnt)	PCT
0	..._amd_vector_fill32float32float32	1	233336	233336	233336	1
1	void benchmark_func000_vector_typeFl...	1	180000	180000	180000	0
2	void benchmark_func001_vector_typeFl...	1	179040	179040	179040	0
3	void benchmark_func002_vector_typeFl...	1	179200	179200	179200	0
4	void benchmark_func003_vector_typeFl...	1	402000	402000	402000	1
5	void benchmark_func004_vector_typeFl...	1	179360	179360	179360	0
6	void benchmark_func005_vector_typeFl...	1	179680	179680	179680	0
7	void benchmark_func006_vector_typeFl...	1	179680	179680	179680	0
8	void benchmark_func007_vector_typeFl...	1	180000	180000	180000	0
9	void benchmark_func008_vector_typeFl...	1	180000	180000	180000	0

ANALYSIS



CLI Analyzer

GUI

CSV Suite

CLI

PROFILING

Perfmon Counters

ROC Profiler

Microbench

MI GPU 1

MI GPU 2

MI GPU 3

...

MI GPU N



Client-side tool

Client-side tool

Modes

Modes change the fundamental behavior of the Omnipperf command line tool.

1. Profile

- Toggle Roofline
- Filter performance counters

2. Analyze

- Generate data tables directly in terminal
- Launch a webpage to visualize

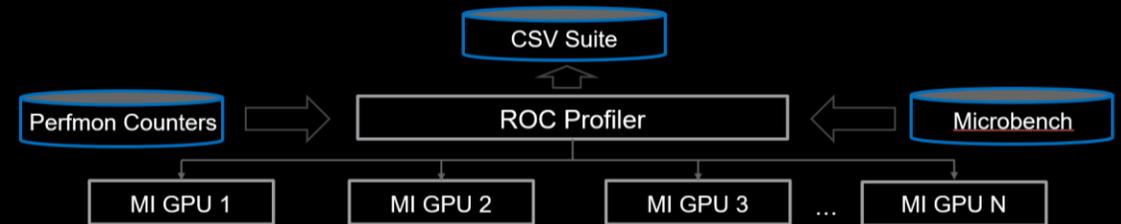
```
colramos@sv-pdp-2:~$ omniperf profile --help
usage:
omniperf profile --name <workload_name> [profile options] [roofline options] -- <profile_cmd>

-----

Examples:

omniperf profile -n vcopy_all -- ./vcopy 1048576 256
omniperf profile -n vcopy_SPI_TD -b SQ TCC -- ./vcopy 1048576 256
omniperf profile -n vcopy_kernel -k vecCopy -- ./vcopy 1048576 256
omniperf profile -n vcopy_disp -d 0 -- ./vcopy 1048576 256
omniperf profile -n vcopy_roof --roof-only -- ./vcopy 1048576 256

-----
```



Client-side tool

Modes

Modes change the fundamental behavior of the Omnipperf command line tool.

1. Profile

- Toggle Roofline
- Filter performance counters

2. Analyze

- Generate data tables directly in terminal
- Launch a webpage to visualize

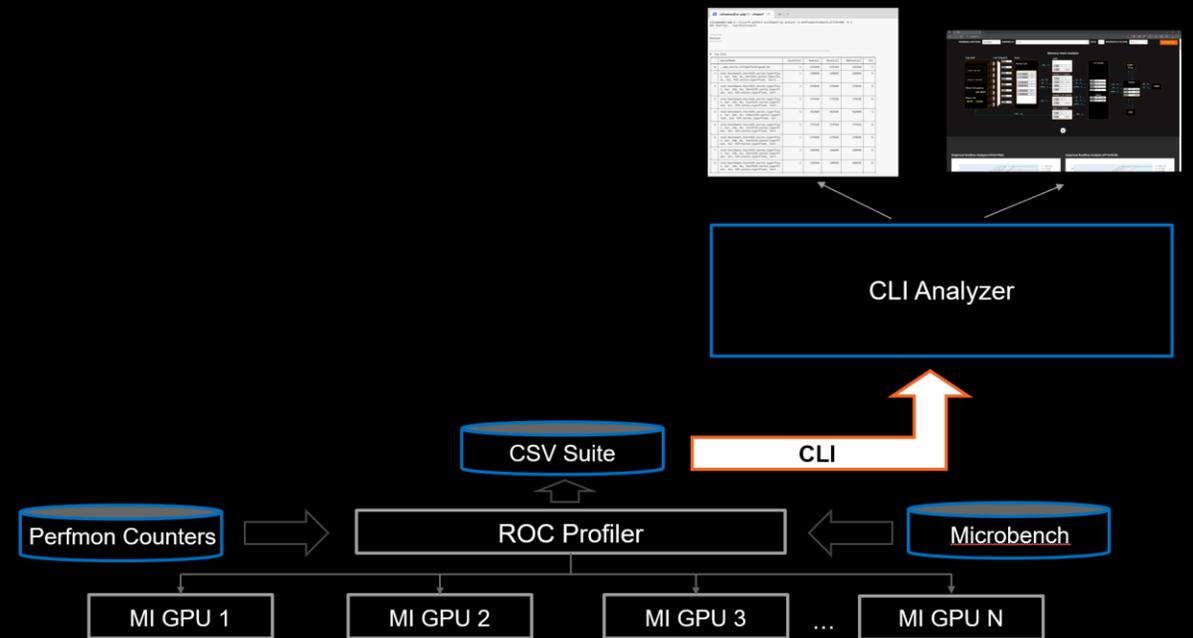
```
colramos@sv-pdp-2:~$ omniperf analyze --help
usage:
omnipperf analyze --path <workload_path> [analyze options]

-----

Examples:

omnipperf analyze -p workloads/vcopy/mi200/ --list-metrics gfx90a
omnipperf analyze -p workloads/mixbench/mi200/ --filter-dispatch-ids 12 34 --decimal 3
omnipperf analyze -p workloads/mixbench/mi200/ --gui

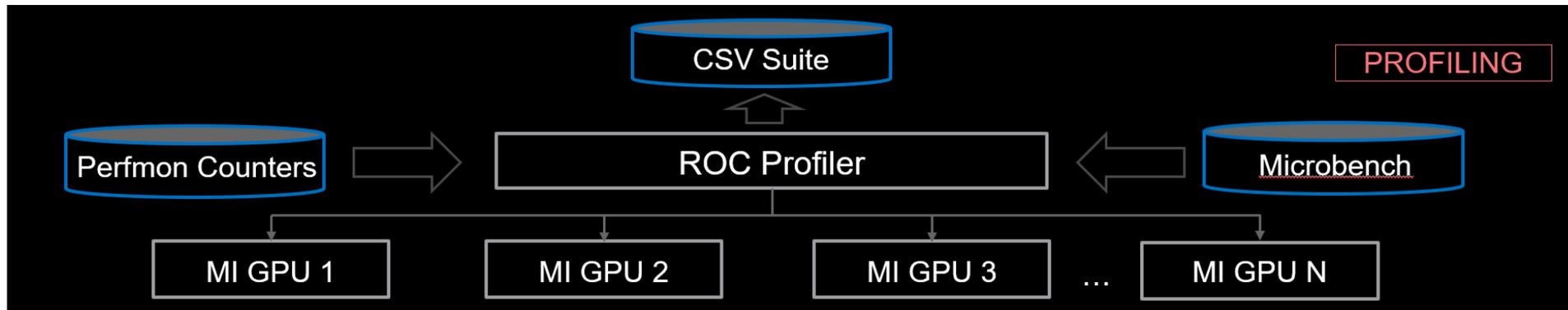
-----
```





“Modes” in Omnipperf

Profile Mode



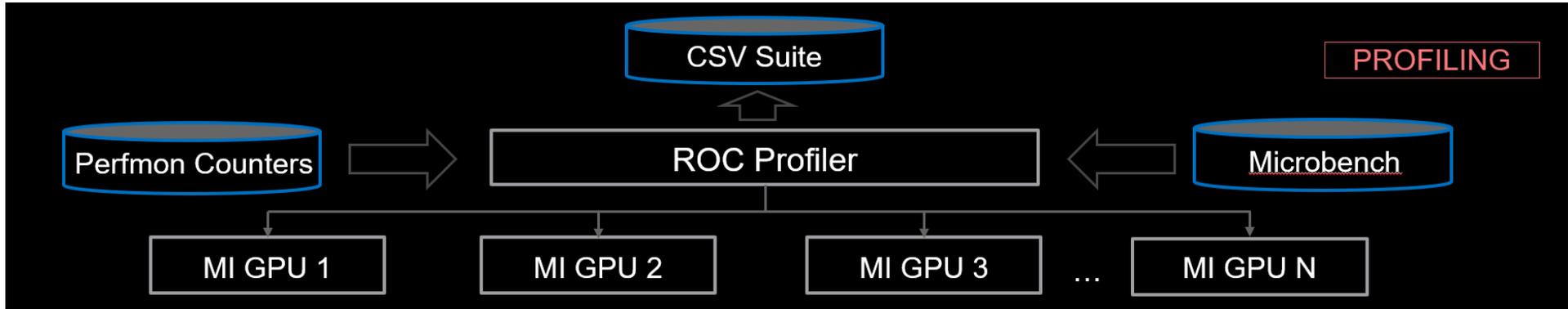
Features:

- Runtime Filtering

`--kernel`, `--ipblocks`, `--dispatch`

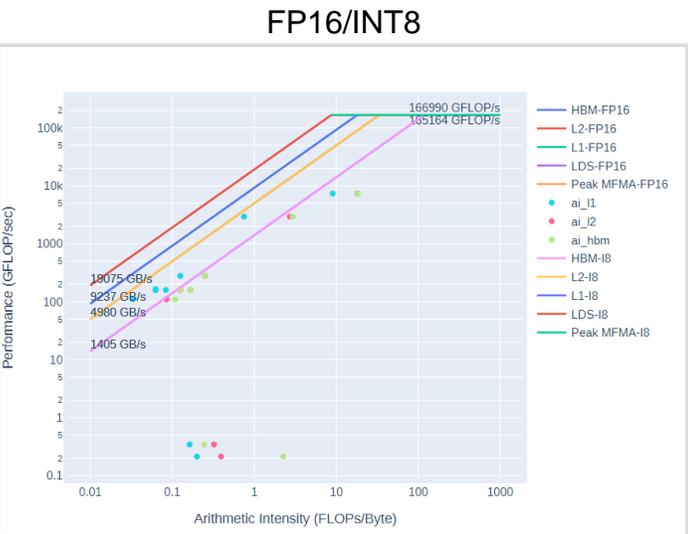
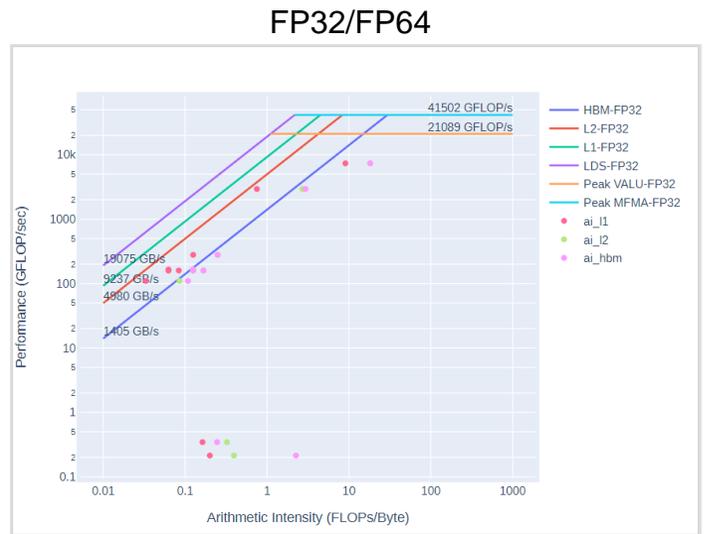
- The `-k` `<kernel>` flag allows for kernel filtering, which is compatible with the current rocpf utility.
- The `-d` `<dispatch>` flag allows for dispatch ID filtering, which is compatible with the current rocpf utility.
- The `-b` `<ipblocks>` allows system profiling on one or more selected IP blocks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

Profile Mode



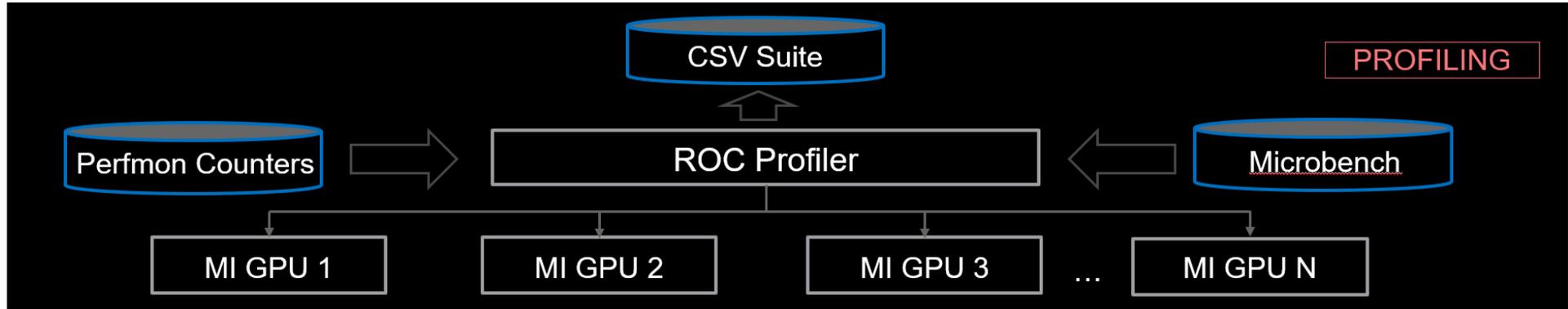
Features:

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



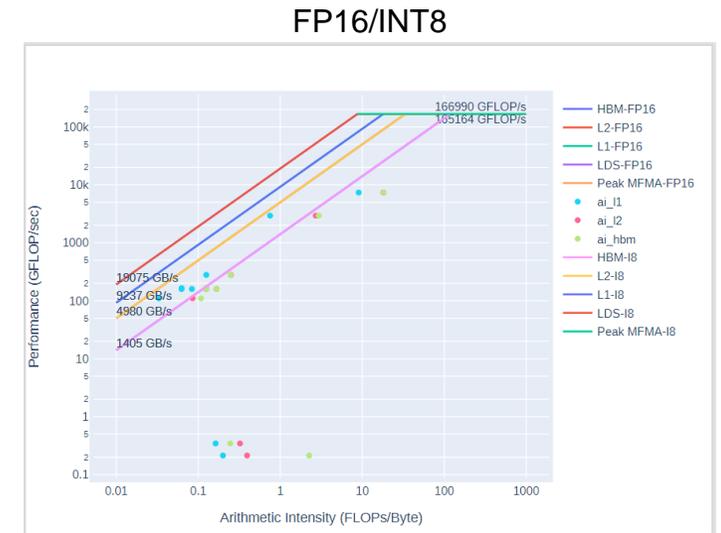
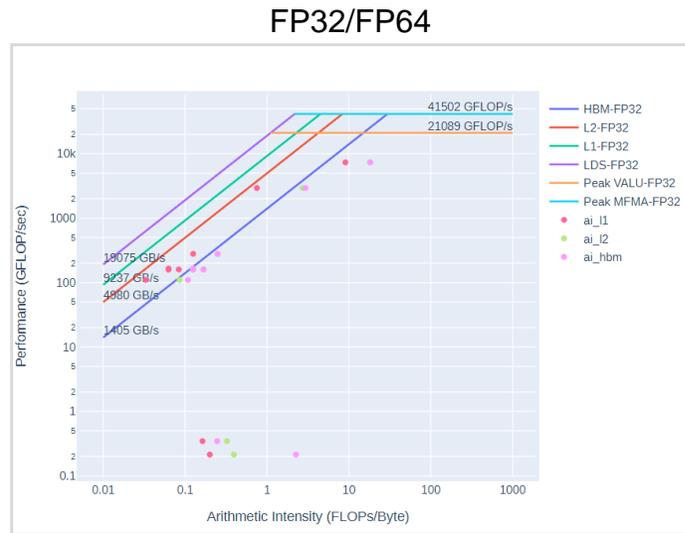
The above plots are saved as PDF output when the --roof-only option is used

Profile Mode



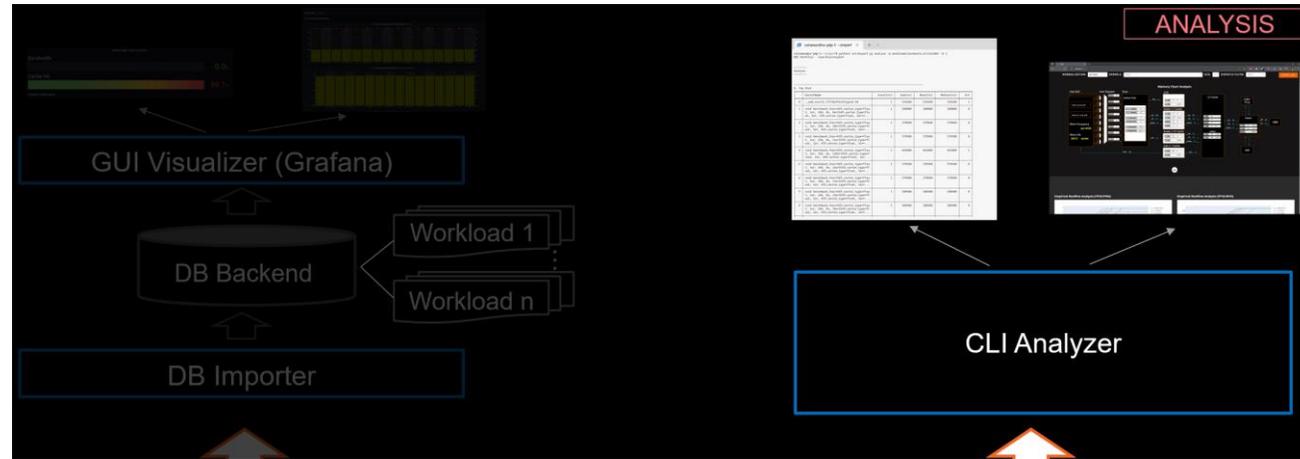
Features:

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



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

Analyze Mode



Features:

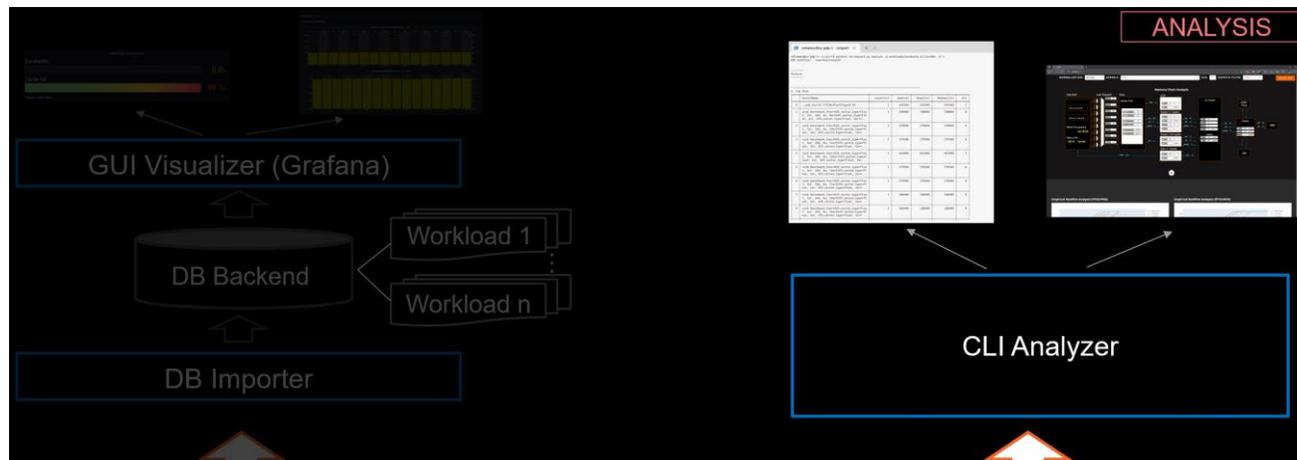
- List top kernels or view list of metrics
`--list-kernels`, `--list-metrics`

```
colramos@sv-pdp-2:~/GitHub/omnipperf-pub$ ./src/omnipperf 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_vector
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_vector
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/omnipperf-pub$ ./src/omnipperf 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
```

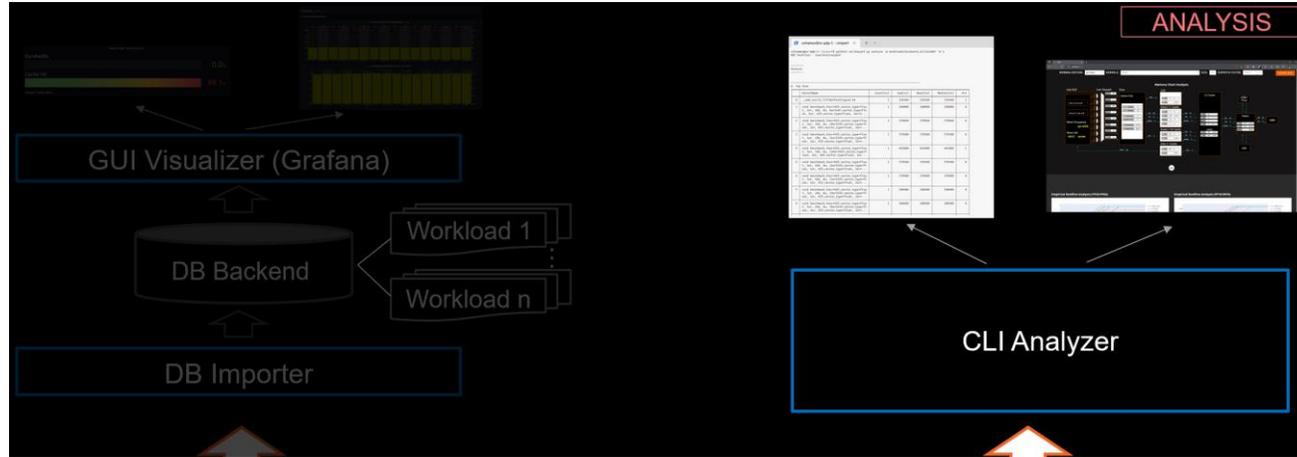
Analyze

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`

```
colranos@sv-pdp-2:~/Github/omniperv-pub$ ./src/omniperv analyze -p workloads/mix_all/mi200/ --metric 5
```

Analyze

0. Top Stat

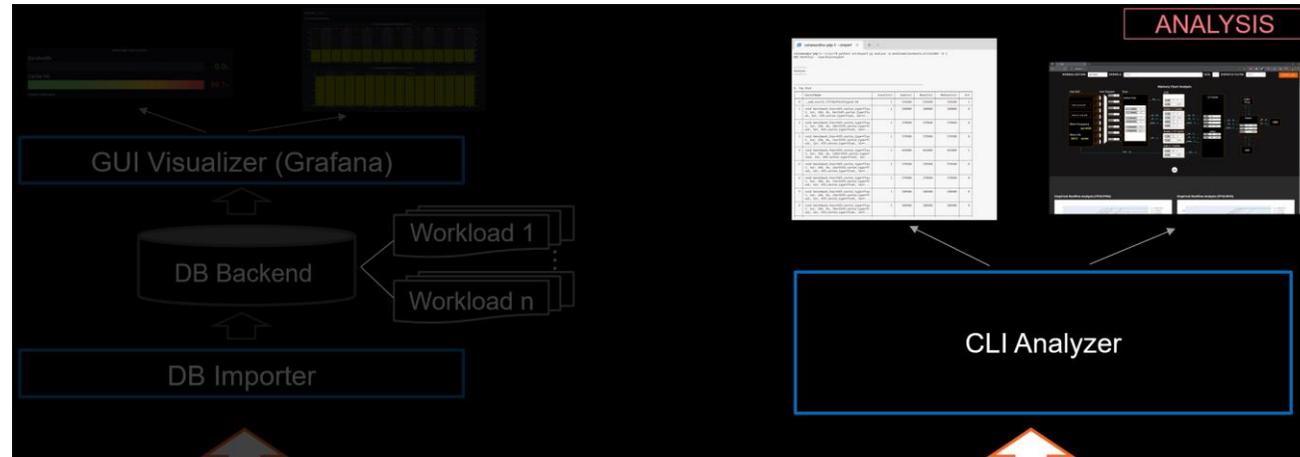
	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]	1	3353042.00	3353042.00	3353042.00	7.87
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vector_type<float, 2u>...)	1	1721239.00	1721239.00	1721239.00	4.04
2	void benchmark_func<double, 256, 8u, 512u>(double, double*) [clone .kd]	1	1718840.00	1718840.00	1718840.00	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:

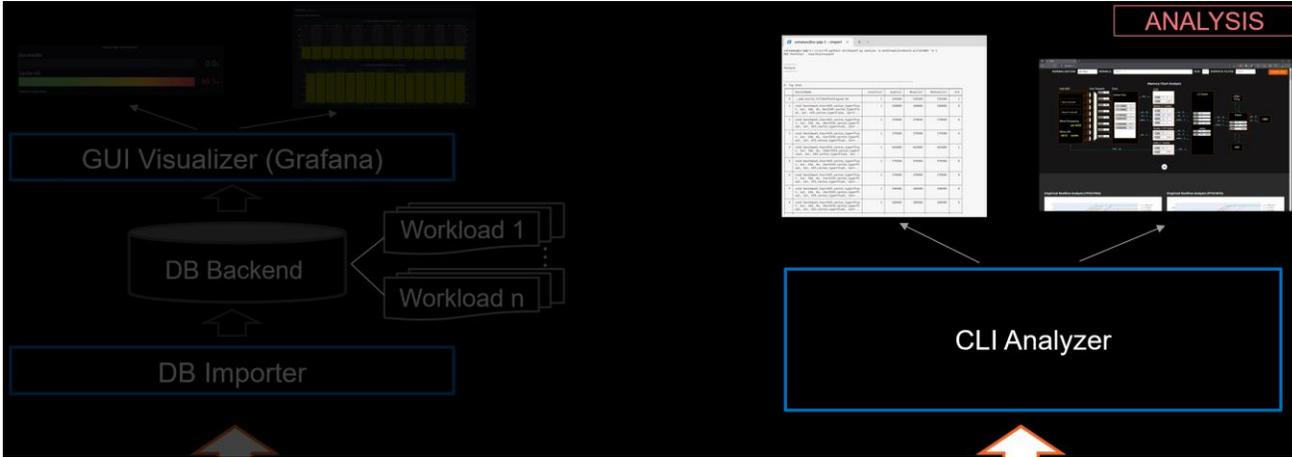
- List top kernels or view list of metrics
`--list-kernels, --list-metrics`
- Filter available kernels, dispatches, gpu-ids
`--kernel, --dispatch, --gpu-id`
- Filter by metric id(s)
`--metric`
- Change normalization unit, time unit, or decimal
`--normal-unit, --time-unit, --decimal`

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:

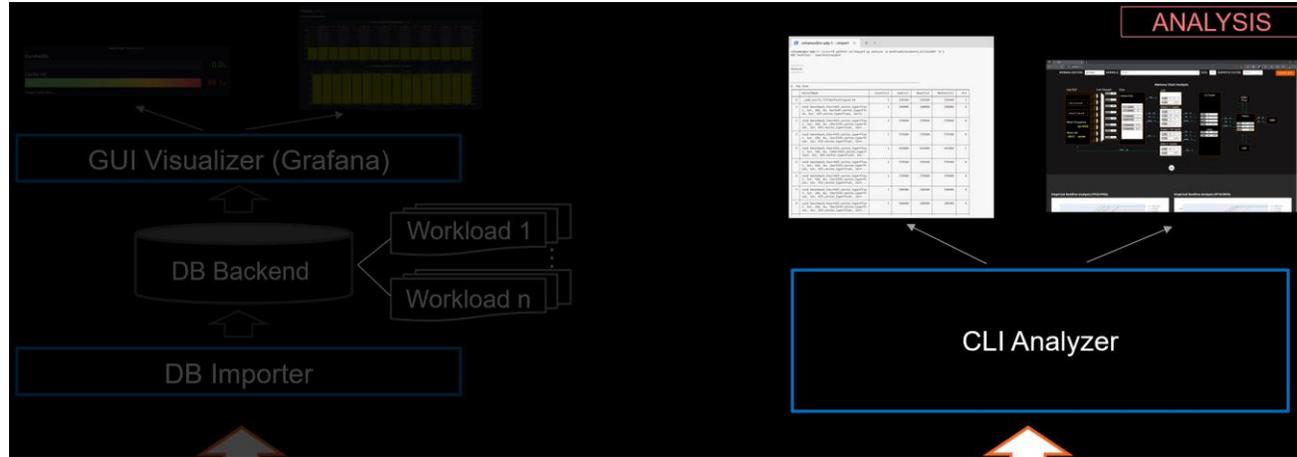
- Baseline Analysis

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

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%)	33.16988268071795	0.0 (-100.0%)
2.1.1	VALU IOPs	2326.1937250893497	398.91 (-82.85%)	GIop	22630.4	22630.4 (0.0%)	10.279865880449968	1.76 (-82.85%)
2.1.2	MFMA FLOPs (BF16)	0.0	0.0 (nan%)	GFlop	98521.6	98521.6 (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%)	0.0 (nan%)	Conflicts/access	32	32.0 (0.0%)	0.0	0.0 (nan%)
2.1.15	Instr Cache Hit Rate	99.99239808071251	99.91 (-0.88%)	Pct	100	100.0 (0.0%)	99.99239808071251	99.91 (-0.88%)
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	20.35928143712575	50.0 (145.59%)	Pct	100	100.0 (0.0%)	20.35928143712575	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	3.814906711045504	35.21 (822.95%)	Pct	100	100.0 (0.0%)	3.814906711045504	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%)	13.295128318560454	0.0 (-100.0%)
2.1.28	Instr Fetch Latency	18.298147264262635	21.37 (16.76%)	Cycles		0.0 (nan%)		0.0 (nan%)

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
5. Command Processor (CPC/CPF)								
5.1 Command Processor Fetcher								
Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit

Analyze Mode (cont.)



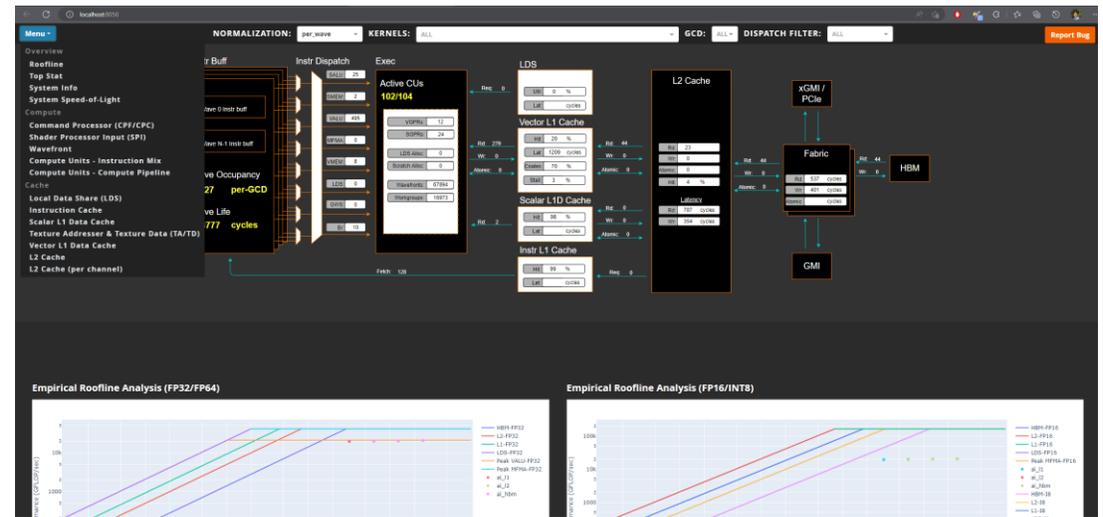
Features:

- 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

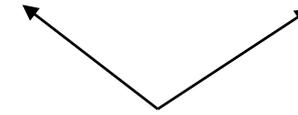
Analyze Mode - Methodology

- Everything we do in analyze mode is built on top of “yml config files”
- These dynamic config files are flexible and can be easily customized
- The configs are used in both CLI and standalone GUI

```
coltanos@sv-pdp-2--3$ omniperf analyze -p workload/mix_all/v1206/ -b 2
```

Analyze

KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0 void benchmark_func_int, 256, 8u, 512u<int, int> [clone .ld]	1	3353042.00	3353042.00	3353042.00	7.87
1 void benchmark_func_HIP_vector_typefloat, 2u, 256, 8u, 512u<HIP_vector_typefloat, 2u, HIP_vector_typefloat, 2u, ...	1	1721239.00	1721239.00	1721239.00	4.04
2 void benchmark_func_double, 256, 8u, 512u<double, double> [clone .ld]	1	1718040.00	1718040.00	1718040.00	4.02
3 void benchmark_func_int, 256, 8u, 256u<int, int> [clone .ld]	1	1693880.00	1693880.00	1693880.00	3.98
4 void benchmark_func_half2, 256, 8u, 512u<_half2, _half2> [clone .ld]	1	1678021.00	1678021.00	1678021.00	3.92
5 void benchmark_func_float, 256, 8u, 512u<float, float> [clone .ld]	1	1661402.00	1661402.00	1661402.00	3.90
6 void benchmark_func_HIP_vector_typefloat	1	881739.00	881739.00	881739.00	2.07



```
1 ---
2 # Add description
3 # So it could be
4 Metric Descriptio
5 #
6 # Define the pane
7 Panel Config:
8 id: 500
9 title: Command
10 data source:
11 - metric_tab
12 id: 501
13 title: Cr
14 header:
15 metric: Metric
16 avg: Avg
17 min: Min
18 max: Max
19 unit: Unit
20 tips: Tips
21 metric:
22 GPU Busy Cycles:
23 avg: AVG(GRBM_GUI_ACTIVE)
24 min: MIN(GRBM_GUI_ACTIVE)
25 max: MAX(GRBM_GUI_ACTIVE)
26 unit: Cycles/Kernel
27 tips:
28 CPF Busy:
29 avg: AVG(CPF_CPF_STAT_BUSY)
30 min: MIN(CPF_CPF_STAT_BUSY)
31 max: MAX(CPF_CPF_STAT_BUSY)
32 unit: Cycles/Kernel
33 tips:
34 CPF Util:
35 avg: AVG((((100 * CPF_CPF_STAT_BUSY) / (CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE))
36 if ((CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE) != 0) else None))
37 min: MIN((((100 * CPF_CPF_STAT_BUSY) / (CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE))
38 if ((CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE) != 0) else None))
39 max: MAX((((100 * CPF_CPF_STAT_BUSY) / (CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE))
40 if ((CPF_CPF_STAT_BUSY + CPF_CPF_STAT_IDLE) != 0) else None))
41 unit: pct
42 tips:
43 CPF Stall:
44 avg: AVG((((100 * CPF_CPF_STAT_STALL) / CPF_CPF_STAT_BUSY) if (CPF_CPF_STAT_BUSY
45 != 0) else None))
46 min: MIN((((100 * CPF_CPF_STAT_STALL) / CPF_CPF_STAT_BUSY) if (CPF_CPF_STAT_BUSY
47 != 0) else None))
48 max: MAX((((100 * CPF_CPF_STAT_STALL) / CPF_CPF_STAT_BUSY) if (CPF_CPF_STAT_BUSY
49 != 0) else None))
50 unit: Cycles/Kernel
51 tips:
52 L2Cache Intf Busy:
53 avg: AVG(CPF_CPF_TCIU_BUSY)
54 min: MIN(CPF_CPF_TCIU_BUSY)
55 max: MAX(CPF_CPF_TCIU_BUSY)
56 unit: Cycles/Kernel
```

yml configs



Omniperf Metrics

High level Metrics

- System Info

System Info	
Metric	Value
Date	Tue Nov 22 18:11:51 2022 (UTC)
App Command	./test_gemm_bf16 0 1 10000
Host Name	0d0b3c44fd0a
Host CPU	AMD EPYC 7402P 24-Core Processor
Host Distro	Ubuntu 20.04.5 LTS
Host Kernel	5.15.0-52-generic
ROCm Version	5.3.0-63
GFX SoC	mi200
GFX ID	gfx90a
Total SEs	8
Total SQCs	56
Total CUs	104
SIMDs/CU	4
Max Wavefronts Occupancy Per CU	32
Max Workgroup Size	1,024
L1Cache per CU (KB)	16
L2Cache (KB)	8,192
L2Cache Channels	32
Sys Clock (Max) - MHz	1,700
Memory Clock (Max) - MHz	1,600
Sys Clock (Cur) - MHz	800
Memory Clock (Cur) - MHz	1,600
DRAM Bandwidth (GB/s)	1,600

Detailed system info for each app is collected by default

High level Metrics

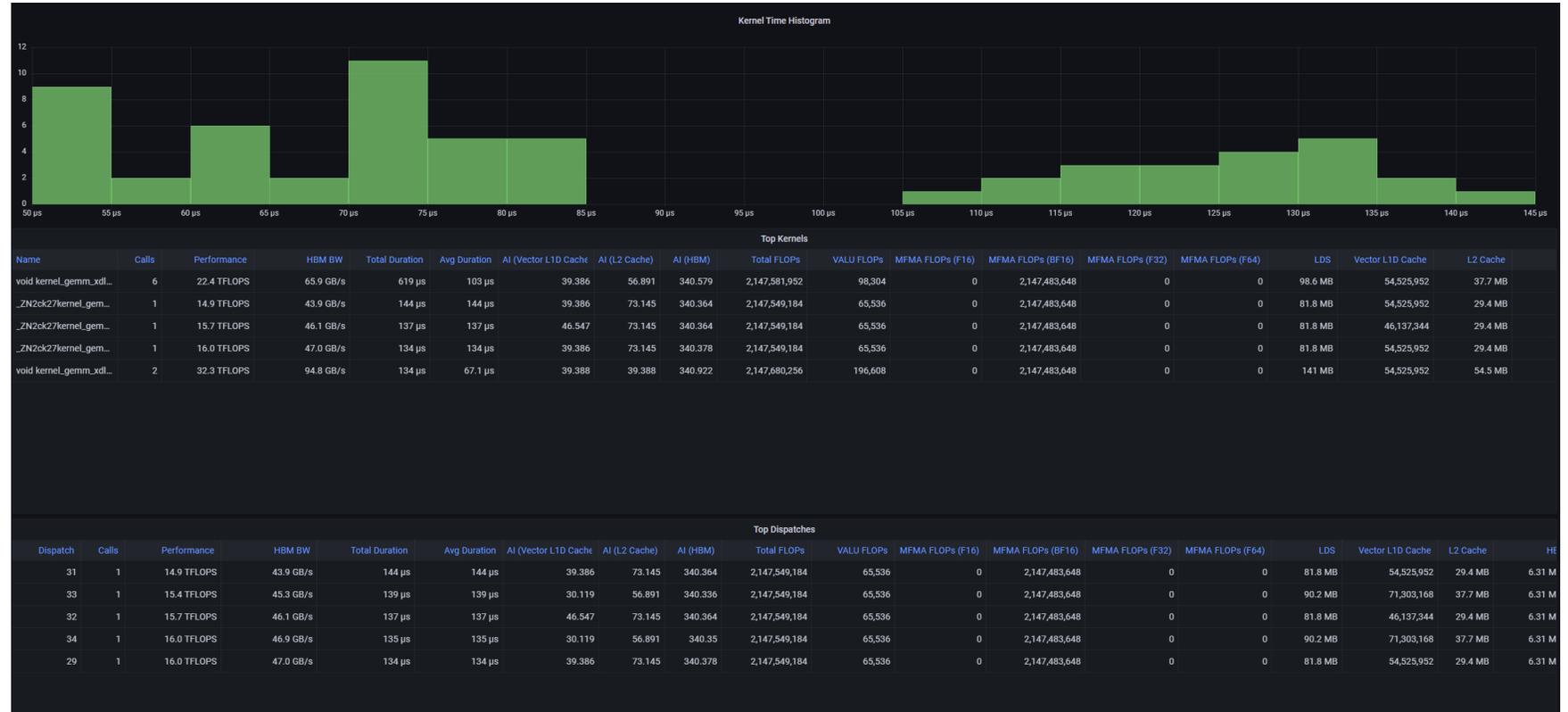
- System Info
- System Speed-of-Light

Speed of Light					Dispatch IDs - Current		Dispatch IDs - Baseline	
Metric	Avg	Unit	Theoretical Max	Pct-of-Peak	Dispatch ID	Kernel Name	Dispatch ID	Kernel Name
VALU FLOPs	2	GFLOP	22,630	0%	0	._ZN2ck27kernel_gemm_xdl_cshuffl...	0	axpy(double*, double*, double*, int)
VALU IOPs	199	GIOP	22,630	1%	1	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA FLOPs (BF16)	27,403	GFLOP	90,522	30%	2	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA FLOPs (F16)	0	GFLOP	181,043	0%	3	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA FLOPs (F32)	0	GFLOP	45,261	0%	4	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA FLOPs (F64)	0	GFLOP	45,261	0%	5	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA IOPs (Int8)	0	GIOP	181,043	0%	6	._ZN2ck27kernel_gemm_xdl_cshuffl...		
Active CUs	63	CUs	104	61%	7	._ZN2ck27kernel_gemm_xdl_cshuffl...		
SALU Util	1	pct	100	1%	8	._ZN2ck27kernel_gemm_xdl_cshuffl...		
VALU Util	5	pct	100	5%	9	._ZN2ck27kernel_gemm_xdl_cshuffl...		
MFMA Util	14	pct	100	14%	10	._ZN2ck27kernel_gemm_xdl_cshuffl...		
VALU Active Threads/Wave	57	Threads	64	89%	11	._ZN2ck27kernel_gemm_xdl_cshuffl...		
IPC - Issue	1	Instr/cycle	5	14%	12	._ZN2ck27kernel_gemm_xdl_cshuffl...		
LDS BW	1,669	GB/sec	22,630	7%	13	._ZN2ck27kernel_gemm_xdl_cshuffl...		
LDS Bank Conflict	0	Conflicts/access	32	1%	14	._ZN2ck27kernel_gemm_xdl_cshuffl...		
Instr Cache Hit Rate	100	pct	100	100%	15	._ZN2ck27kernel_gemm_xdl_cshuffl...		
Instr Cache BW	211	GB/s	6,093	3%	16	void kernel_gemm_xdl_cshuffl_v1<		
Scalar L1D Cache Hit Rate	73	pct	100	73%	17	void kernel_gemm_xdl_cshuffl_v1<		
Scalar L1D Cache BW	3	GB/s	6,093	0%	18	void kernel_gemm_xdl_cshuffl_v1<		
Vector L1D Cache Hit Rate	23	pct	100	23%	19	void kernel_gemm_xdl_cshuffl_v1<		
Vector L1D Cache BW	858	GB/s	11,315	8%	20	void kernel_gemm_xdl_cshuffl_v1<		
L2 Cache Hit Rate	90	pct	100	90%	21	void kernel_gemm_xdl_cshuffl_v1<		
L2-Fabric Read BW	54	GB/s	1,638	3%	22	void kernel_gemm_xdl_cshuffl_v1<		
L2-Fabric Write BW	27	GB/s	1,638	2%	23	void kernel_gemm_xdl_cshuffl_v1<		
L2-Fabric Read Latency	297	Cycles			24	void kernel_gemm_xdl_cshuffl_v1<		
L2-Fabric Write Latency	532	Cycles			25	void kernel_gemm_xdl_cshuffl_v1<		
Wave Occupancy	246	Wavefronts	3,328	7%	26	void kernel_gemm_xdl_cshuffl_v1<		
Instr Fetch BW	0	GB/s	3,046	0%	27	void kernel_gemm_xdl_cshuffl_v1<		
Instr Fetch Latency	53	Cycles			28	void kernel_gemm_xdl_cshuffl_v1<		

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

High level Metrics

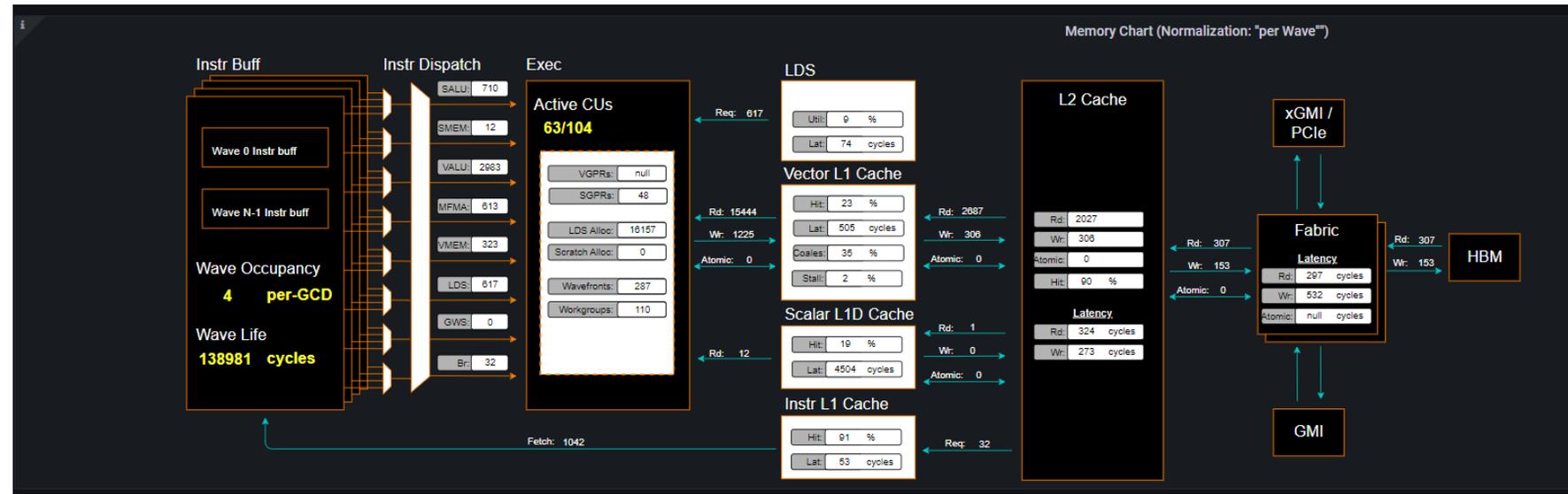
- System Info
- System Speed-of-Light
- Kernel Stats



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

High level Metrics

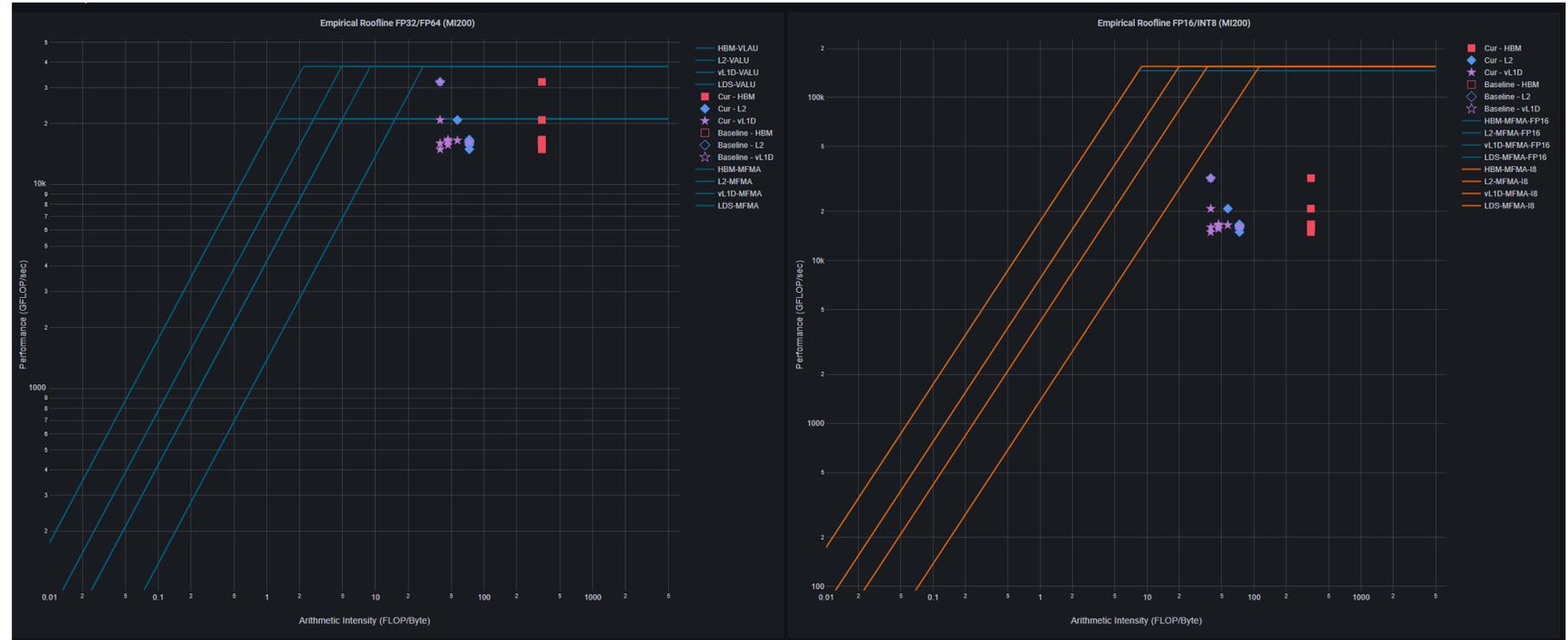
- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis



Illustrate data movement and performance on key components of target architecture

High level Metrics

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



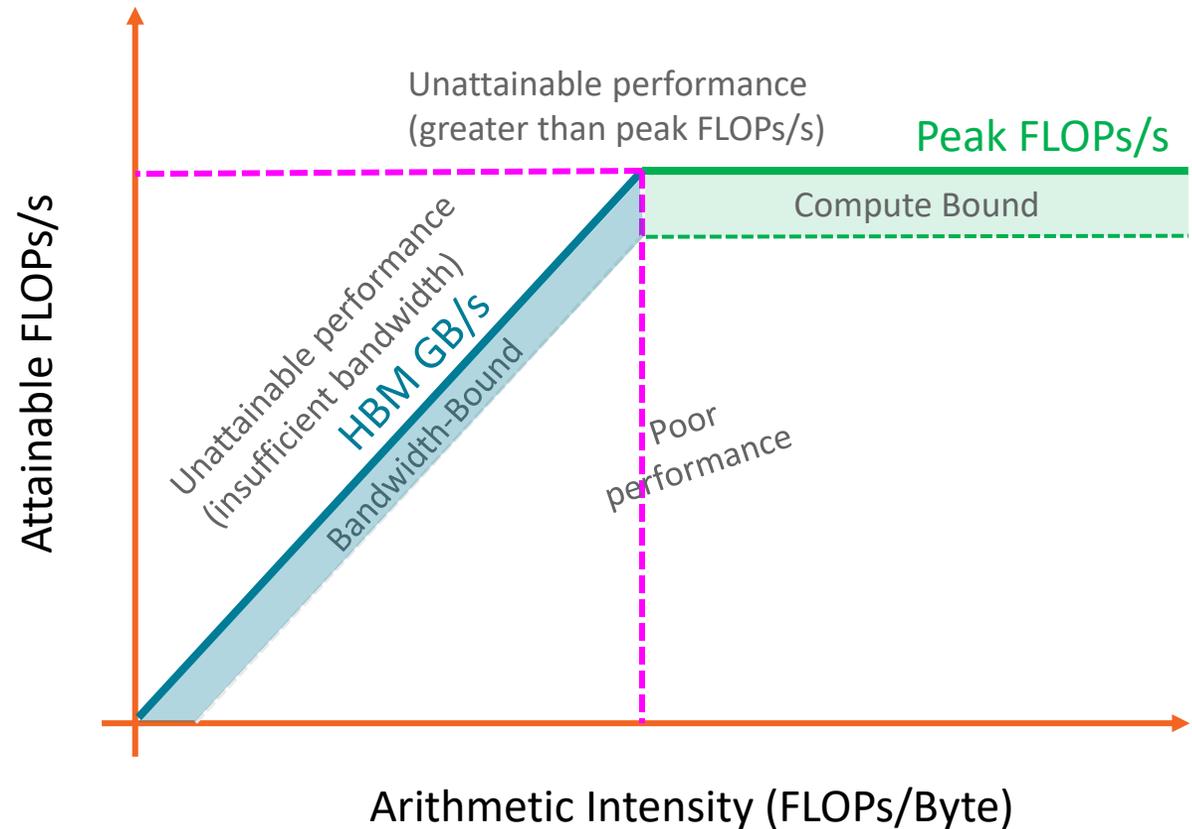
Derived Empirical Roofline analysis broken into two major instruction mixes. Showing application performance relative to measured maximum achievable performance



Roofline Analysis Methodology

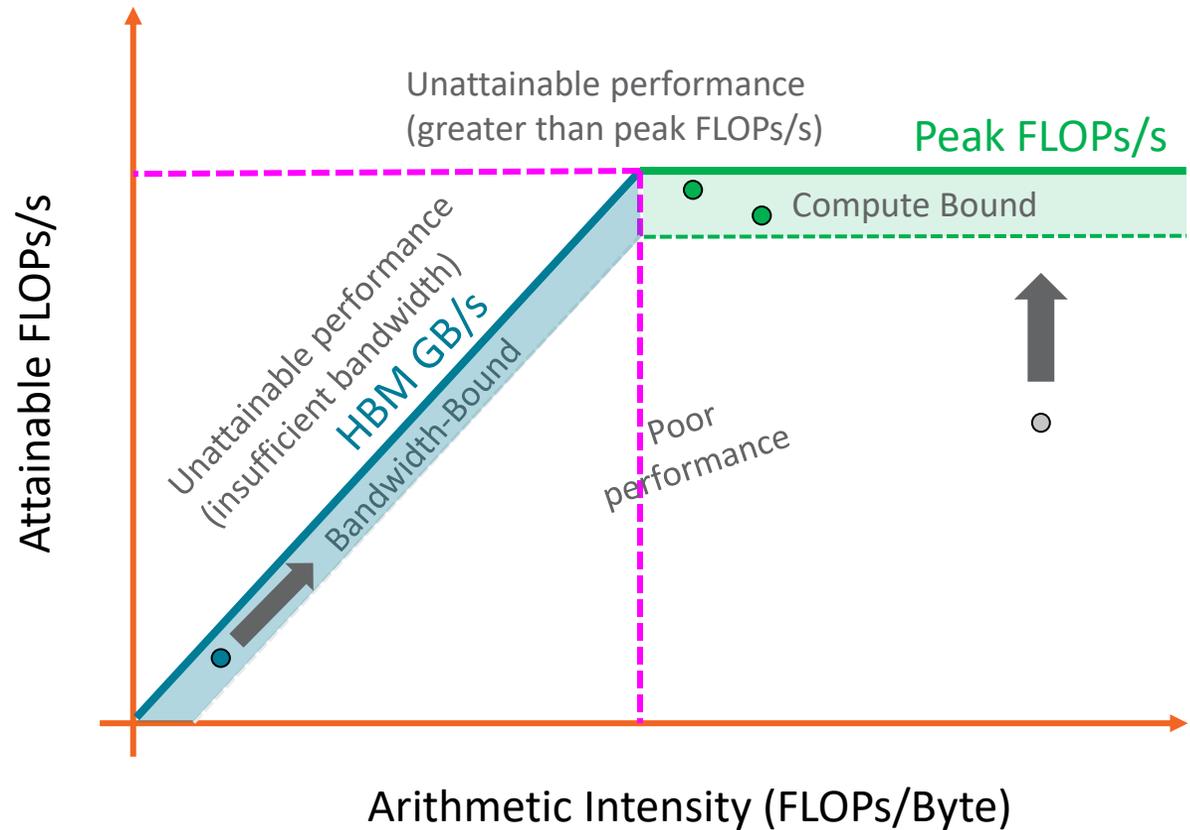
Background – What is roofline?

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

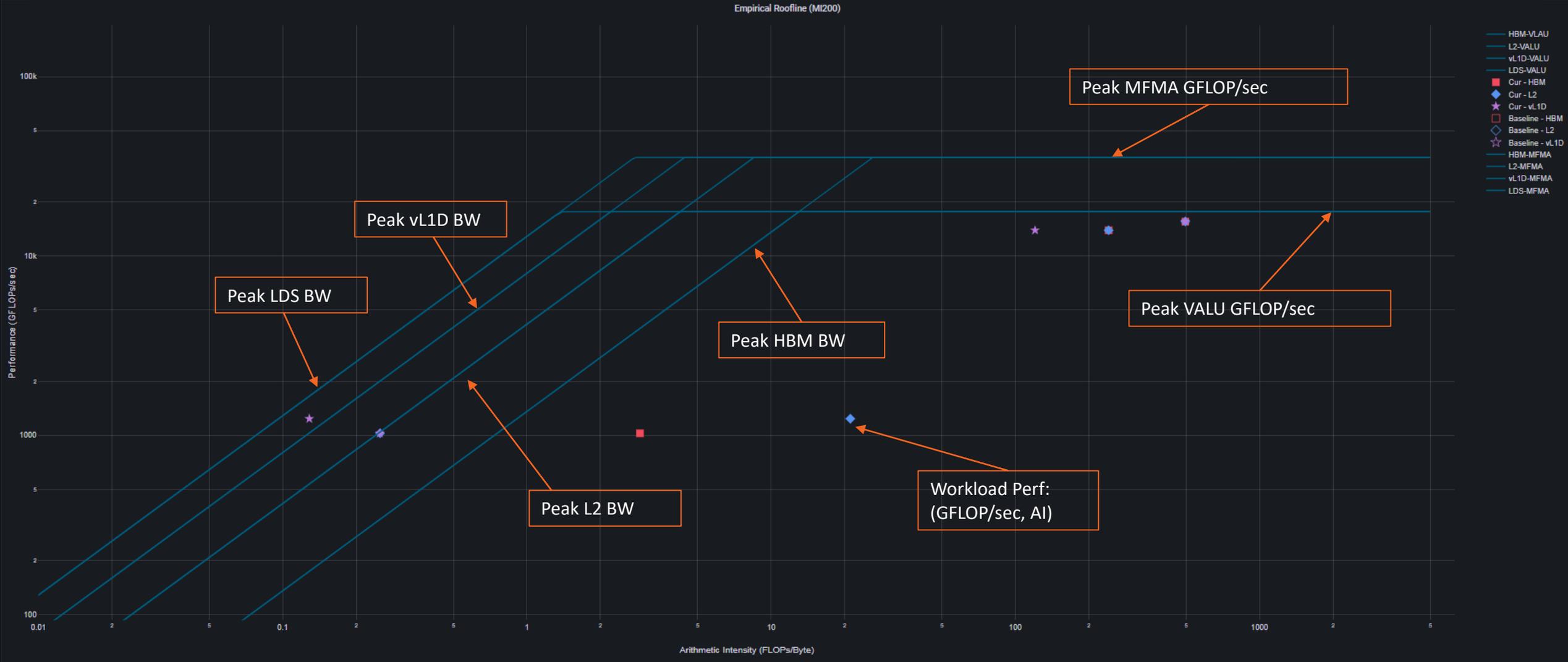


Background – What is roofline?

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



Introduction - Empirical Hierarchical Roofline on AMD Instinct™ MI250X GPU



Empirical Hierarchical Roofline on MI200 - Arithmetic

$$\begin{aligned}
 \text{Total_FLOP} = & 64 * (\text{SQ_INSTS_VALU_ADD_F16} + \text{SQ_INSTS_VALU_MUL_F16} + \text{SQ_INSTS_VALU_TRANS_F16} + 2 * \text{SQ_INSTS_VALU_FMA_F16}) \\
 & + 64 * (\text{SQ_INSTS_VALU_ADD_F32} + \text{SQ_INSTS_VALU_MUL_F32} + \text{SQ_INSTS_VALU_TRANS_F32} + 2 * \text{SQ_INSTS_VALU_FMA_F32}) \\
 & + 64 * (\text{SQ_INSTS_VALU_ADD_F64} + \text{SQ_INSTS_VALU_MUL_F64} + \text{SQ_INSTS_VALU_TRANS_F64} + 2 * \text{SQ_INSTS_VALU_FMA_F64}) \\
 & + 512 * \text{SQ_INSTS_VALU_MFMA_MOPS_F16} \\
 & + 512 * \text{SQ_INSTS_VALU_MFMA_MOPS_BF16} \\
 & + 512 * \text{SQ_INSTS_VALU_MFMA_MOPS_F32} \\
 & + 512 * \text{SQ_INSTS_VALU_MFMA_MOPS_F64}
 \end{aligned}$$

$$\text{Total_IOP} = 64 * (\text{SQ_INSTS_VALU_INT32} + \text{SQ_INSTS_VALU_INT64})$$

$$\text{LDS}_{BW} = 32 * 4 * (\text{SQ_LDS_IDX_ACTIVE} - \text{SQ_LDS_BANK_CONFLICT})$$

$$\text{vL1D}_{BW} = 64 * \text{TCP_TOTAL_CACHE_ACCESSES_sum}$$

$$\begin{aligned}
 \text{L2}_{BW} = & 64 * \text{TCP_TCC_READ_REQ_sum} \\
 & + 64 * \text{TCP_TCC_WRITE_REQ_sum} \\
 & + 64 * (\text{TCP_TCC_ATOMIC_WITH_RET_REQ_sum} + \text{TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum})
 \end{aligned}$$

$$\begin{aligned}
 \text{HBM}_{BW} = & 32 * \text{TCC_EA_RDREQ_32B_sum} + 64 * (\text{TCC_EA_RDREQ_sum} - \text{TCC_EA_RDREQ_32B_sum}) \\
 & + 32 * (\text{TCC_EA_WRREQ_sum} - \text{TCC_EA_WRREQ_64B_sum}) + 64 * \text{TCC_EA_WRREQ_64B_sum}
 \end{aligned}$$

$$AI_{LDS} = \frac{\text{TOTAL_FLOP}}{\text{LDS}_{BW}}$$

$$AI_{vL1D} = \frac{\text{TOTAL_FLOP}}{\text{vL1D}_{BW}}$$

$$AI_{L2} = \frac{\text{TOTAL_FLOP}}{\text{L2}_{BW}}$$

$$AI_{HBM} = \frac{\text{TOTAL_FLOP}}{\text{HBM}_{BW}}$$



* All calculations are subject to change without notice

Low level Metrics

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

Agenda

General Overview

1. Introduction
2. Methodology
3. Implementation
 - Profile Mode
 - Analyze Mode
4. Roofline Analysis
5. Basic Examples

Guided Exercises

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

Guided Exercises: Logistics/Preamble

- 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/OLCF/hip-training-series`, we'll be working in the **Lecture5/OmniperfExamples** subdirectory.
 - The READMEs contain all of what I'm saying and include platform-specific instructions for this training in the top-level directory
 - Cole Ramos and Bob Robey will monitor and answer questions while I'm talking
- We have used a publicly available release candidate of Omniperf to generate output for these slides:
 - <https://github.com/AMDRResearch/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:
 - Omnipperf commands to filter through output for common optimization concerns
 - Some optimization direction given certain Omnipperf 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;
}
```

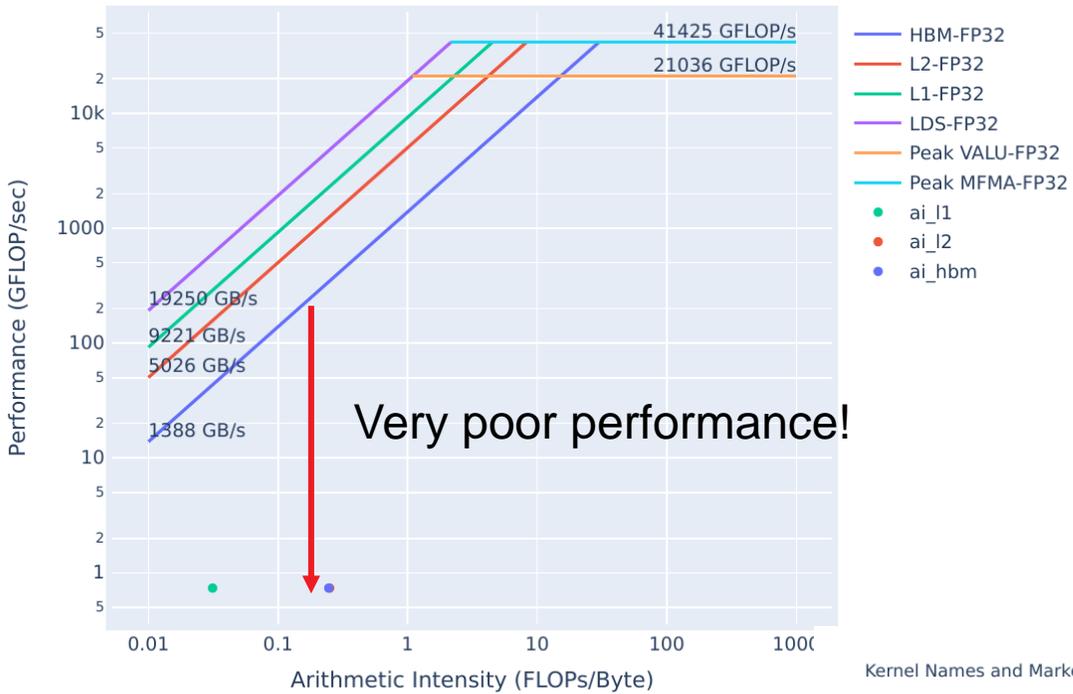
- Where:
 - A is a 1-D array of size n*m
 - x is an array of size m
 - y is an array of size n

Exercise 1: First Things First, Generate a Roofline

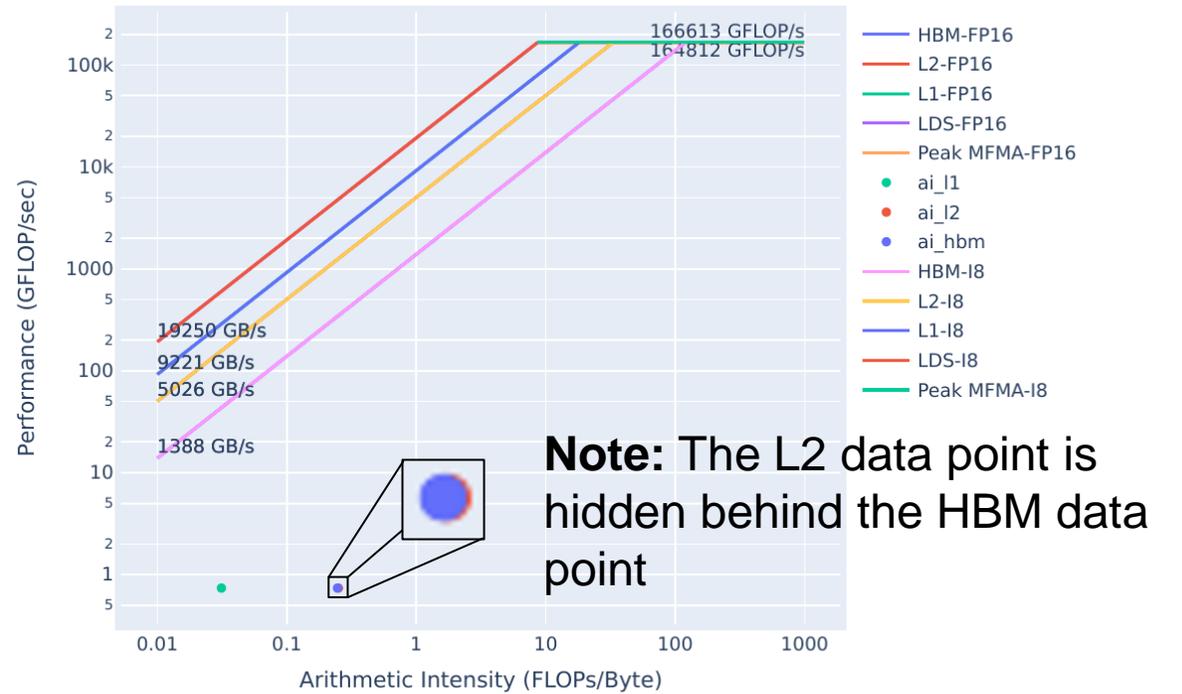
- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
 - `omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
 - The files will appear in the `./workloads/problem_roof_only/mi200` folder.
 - `--roof-only` generates PDF roofline plots, 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

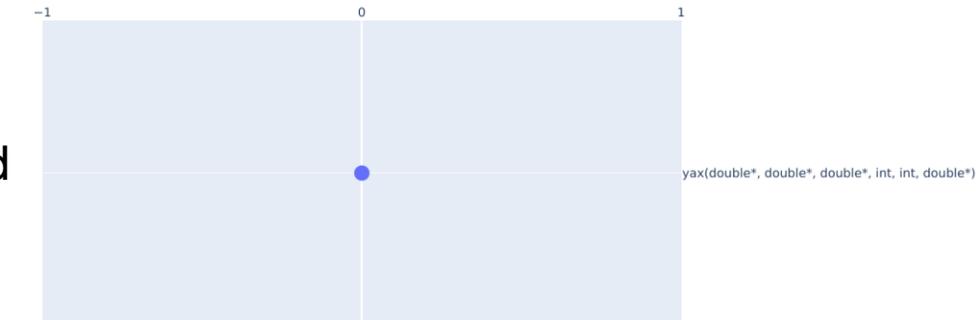
FP32 Roofline Plot



FP16/INT8 Roofline Plot



Kernel legend



Exercise 1: Prep to use Omnipperf 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
- Omnipperf can easily show launch parameter information regardless of the code
 - You just need the dispatch ID
- To generate profiling data, use the commands:
 - `omnipperf profile -n problem --no-roof -- ./problem.exe`
 - `omnipperf 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

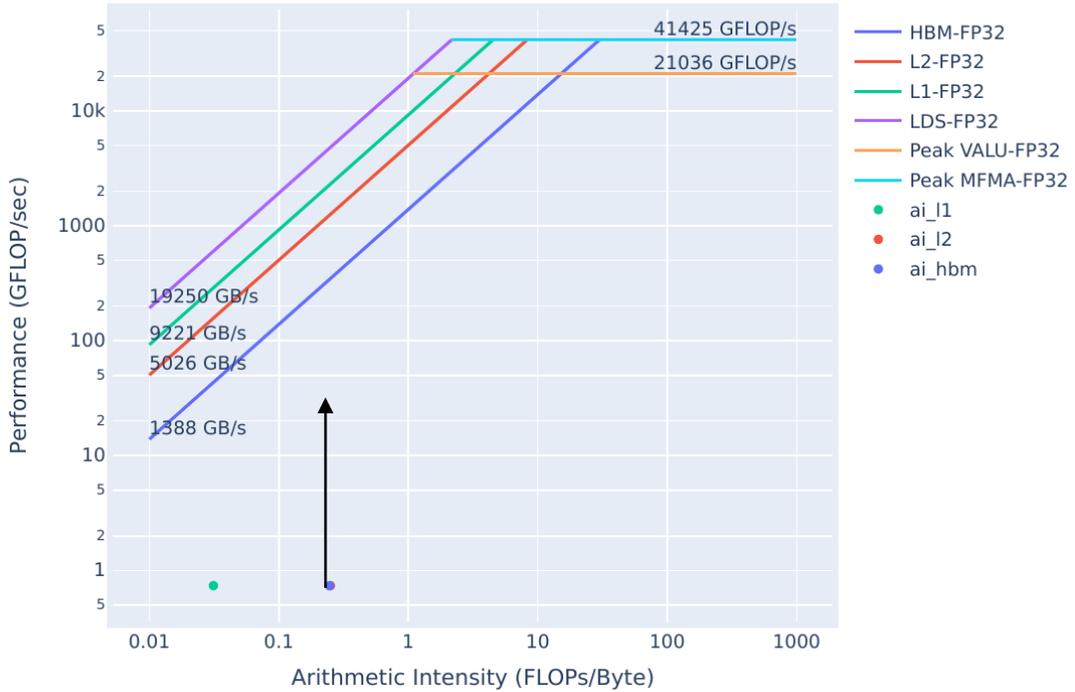
In general, it is difficult to pre-determine optimal launch bounds, so some experimentation is likely necessary

Increased launched wavefronts, which increases Grid Size

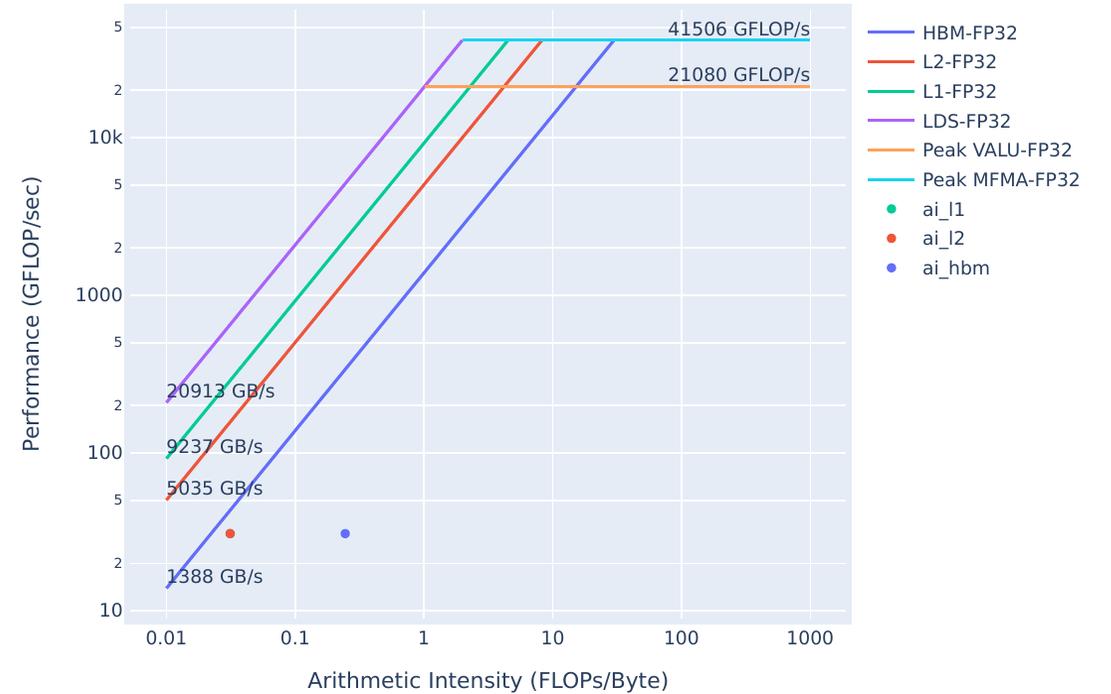
These slides always put **problem** as the baseline, and **solution** as the comparative

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

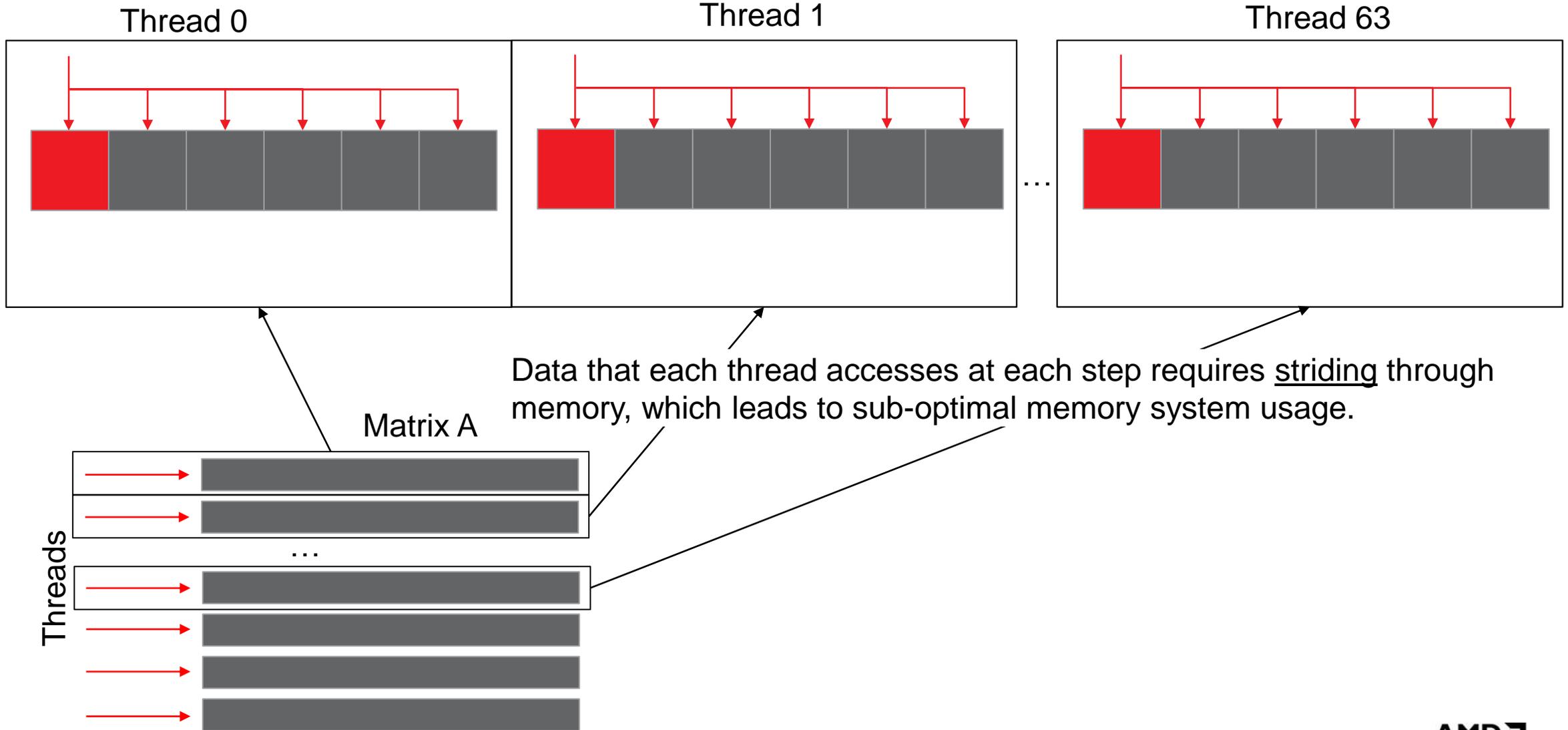
Exercise 3: Register Occupancy Limiter - Takeaways

- 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

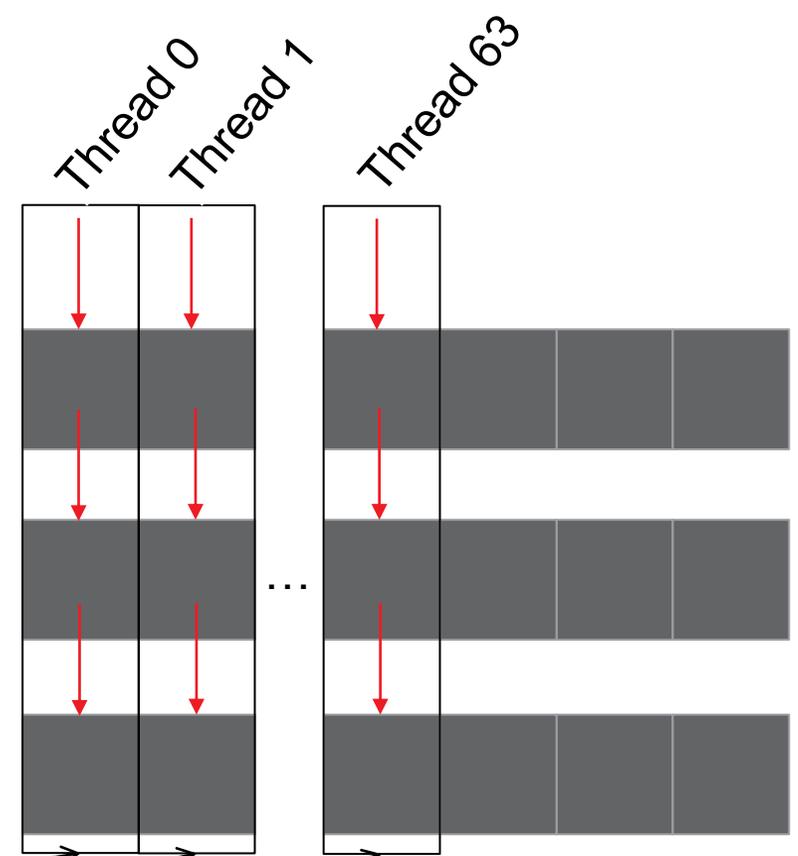
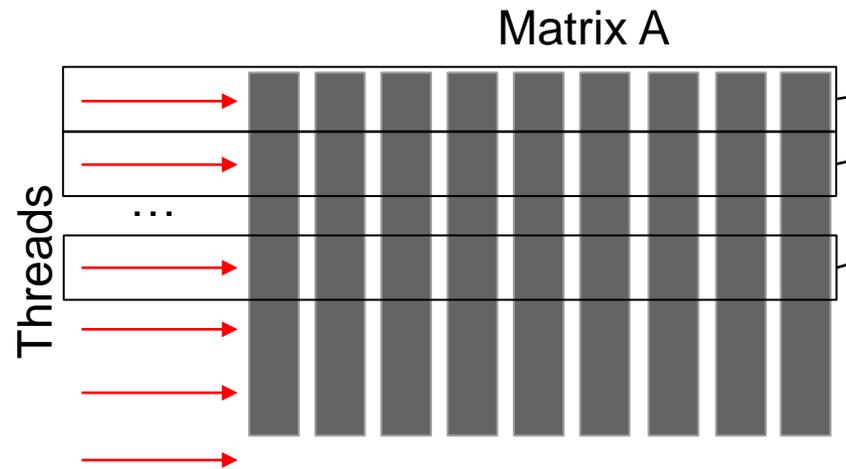
- The way in which threads access memory has a big impact on performance
- “Striding” in global memory has adverse effects on kernel performance, especially on GPUs.
 - “Strided data access patterns” lead to poor utilization of cache memory systems
- These access patterns can be difficult to spot in the code
 - They are valid methods of indexing data
- Using Omnipperf can quickly show if a kernel’s data access is adversarial to the caches

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



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 Omnipperf 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:
 - `omnipperf profile -n problem --no-roof -- ./problem.exe`
 - `omnipperf profile -n solution --no-roof -- ./solution.exe`

Exercise 4: Strided Data Access Pattern – Relevant Omnipperf Output

```
omnipperf 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

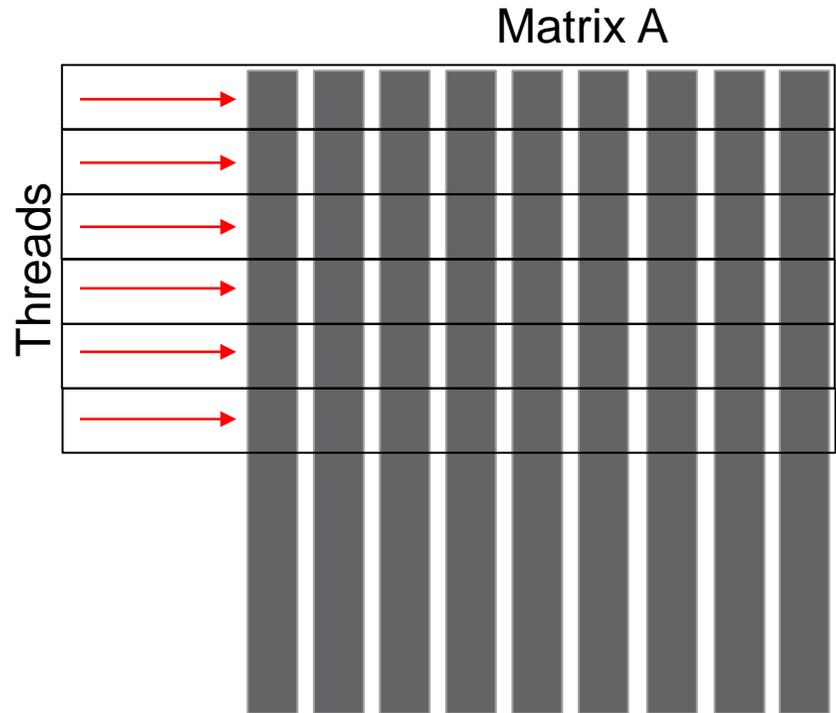
Exercise 4: Omnipperf Speed-of-Light Cache Access Statistics

- This Omnipperf command will show high-level details about L1 and L2 cache accesses:
 - `omnipperf 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

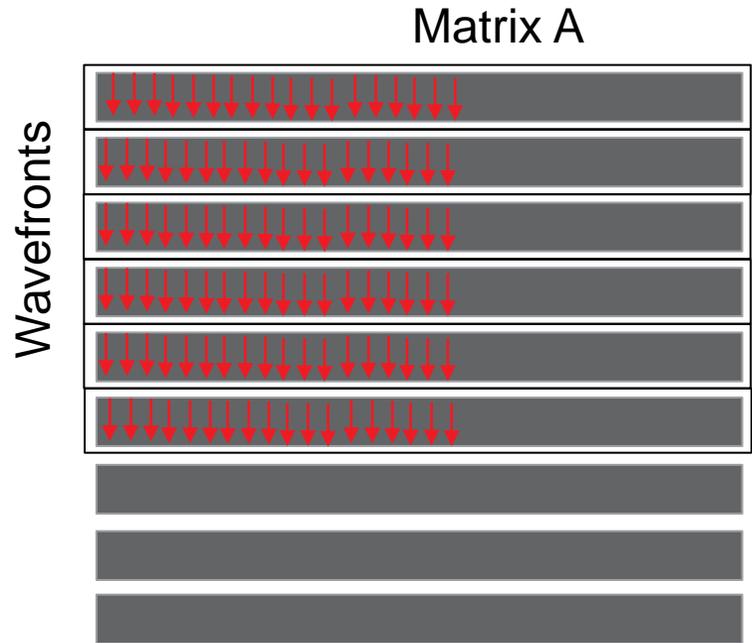
- These types of optimizations are the most difficult to execute
 - Generally, it is difficult to determine if the runtime of one algorithm will be faster than another
- We start with the solution from last exercise as our problem
 - Speed-of-light cache statistics showed that we had ~0% hit rate in 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 Omnipperf 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:
 - `omnipperf profile -n problem --no-roof -- ./problem.exe`
 - `omnipperf 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%)

16. Vector L1 Data Cache
 16.3 L1D Cache Accesses

~29x faster

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
...								
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
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

- ~32x

- ~32x

+ ~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
...								
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
17.3.8	Cache Hit	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	Pct

- ~53x

- ~64x

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

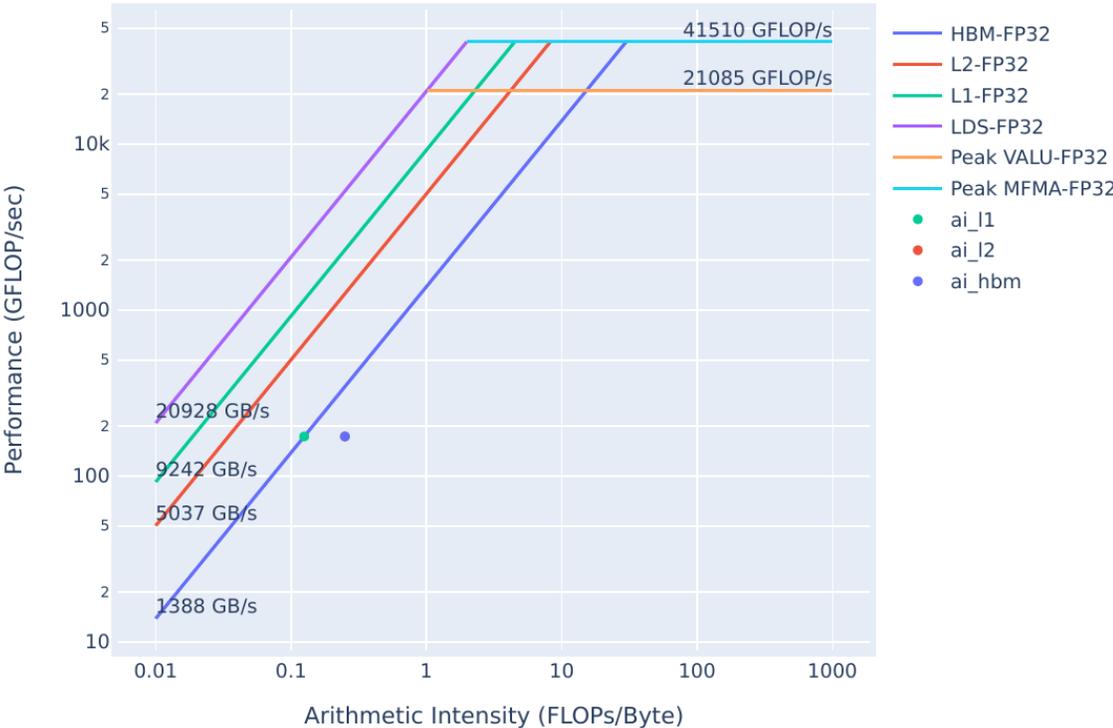
Large relative gain, + ~16% overall



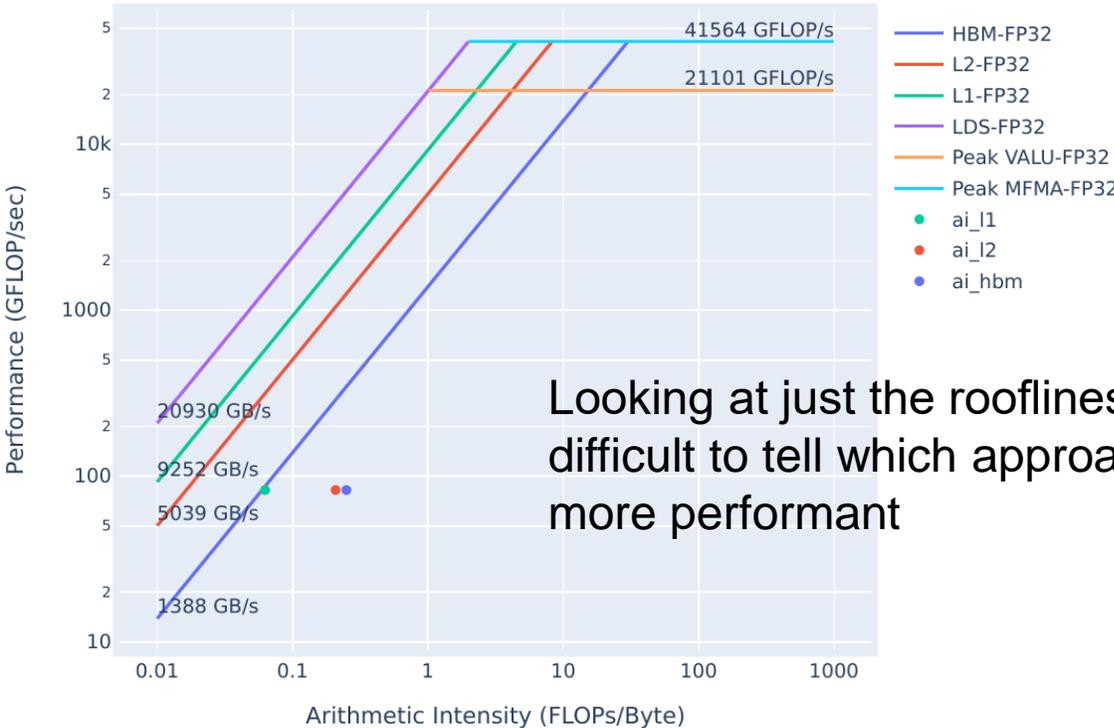
Exercise 5: It Can Be Hard to Compare Rooflines Between Algorithms

- `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: Omnipperf Detailed Cache Statistics - Takeaways

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

DISCLAIMERS AND ATTRIBUTIONS

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

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

© 2023 Advanced Micro Devices, Inc. All rights reserved.

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

AMD 