Introduction to Omniperf

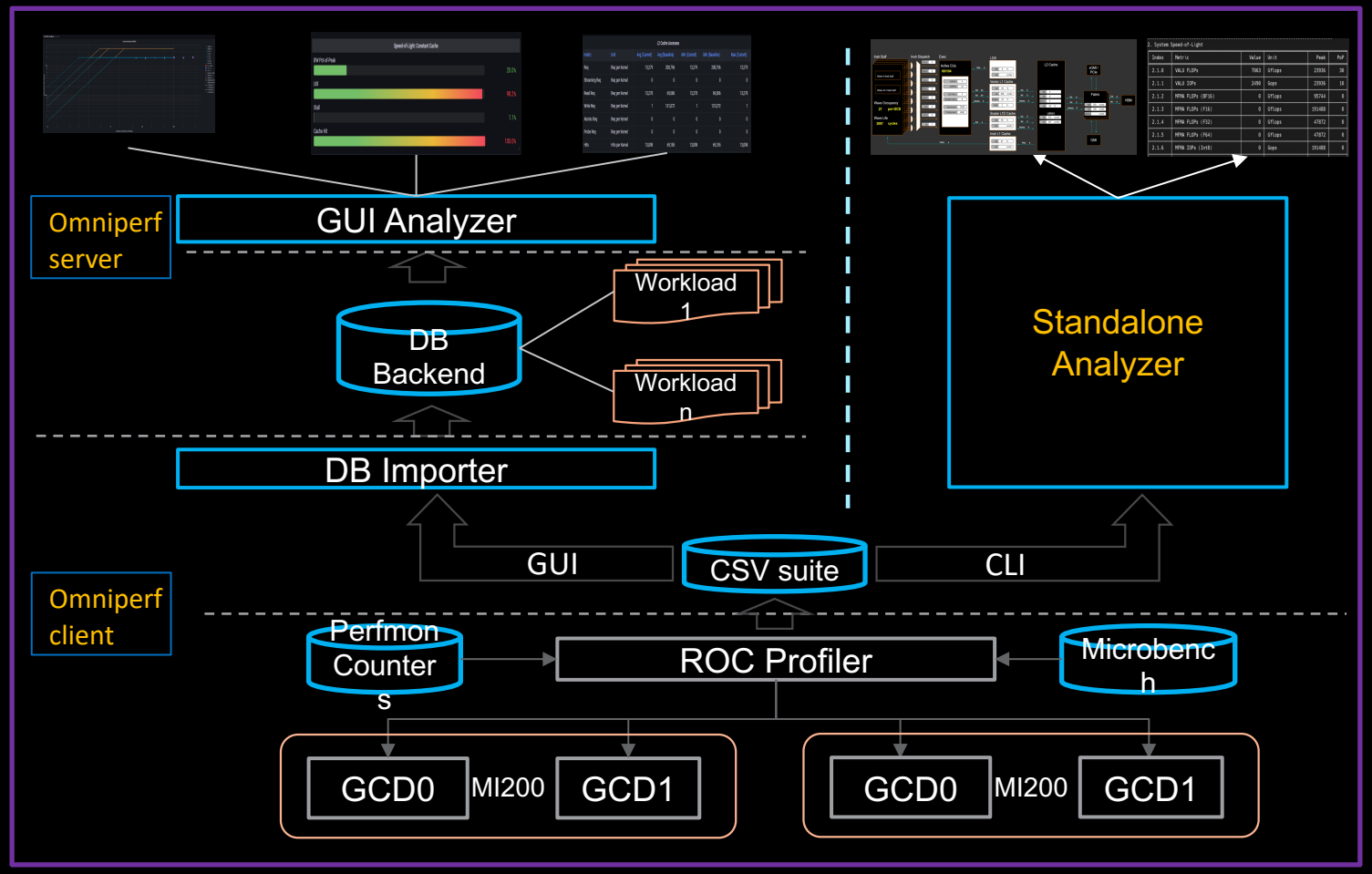
Ian Bogle, Cole Ramos, Suyash Tandon, Xiaomin Lu, Noah Wolfe,
George Markomanolis, Samuel Antao

LUMI Pre-hackathon training
November 22, 2023

AMD
together we advance_

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

Background – AMD Profilers

ROC-profiler (rocprof)

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

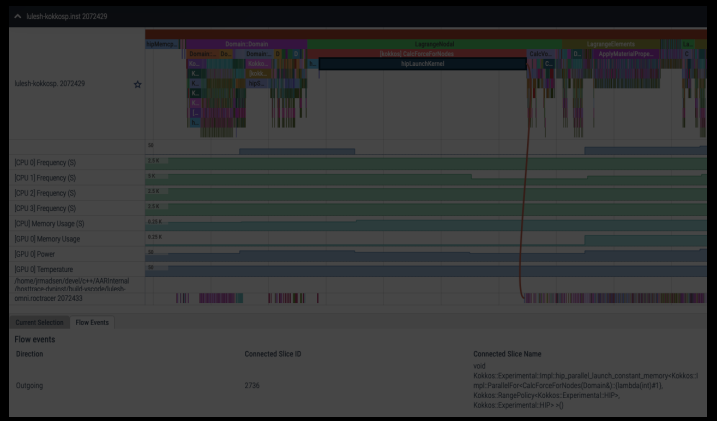
Omnitrace

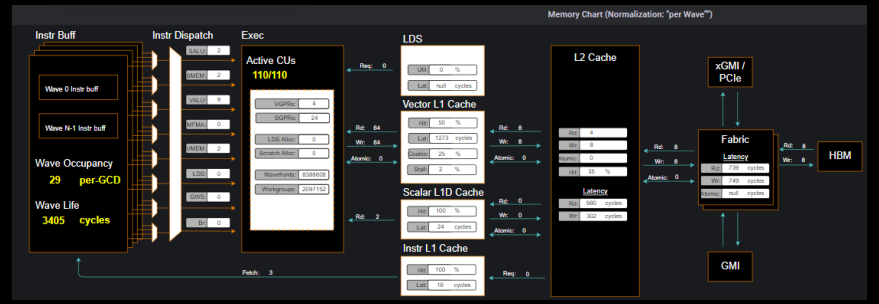
Trace collection	Comprehensive trace collection				
	CPU	GPU			
Supports	CPU copy	HIP API	HSA API	GPU Kernels	
	OpenMP®	MPI	Kokkos	p-threads	multi-GPU
Visualisation	Traces visualized with Perfetto				

Omniperf

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





Client-side installation (if required)

Download the latest version from here: https://github.com/AMDResearch/omniperf/releases

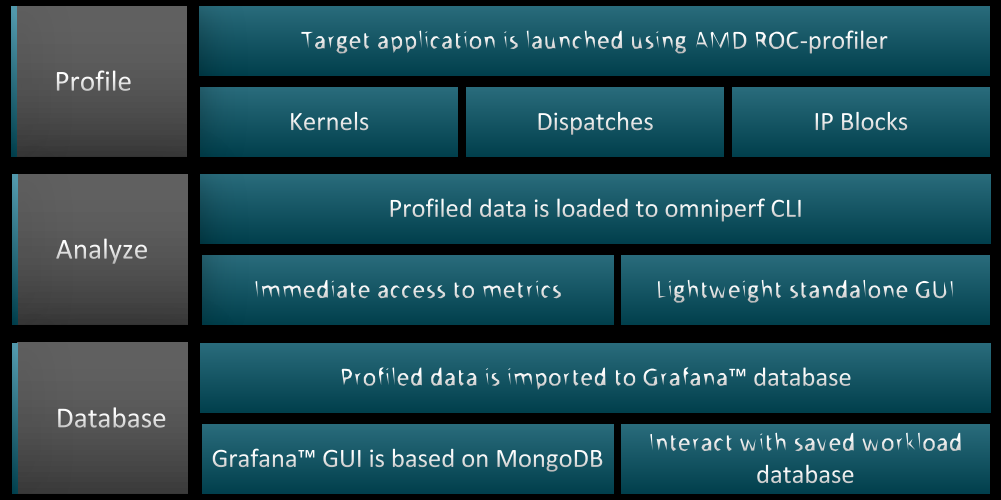
Full documentation: https://amdresearch.github.io/omniperf/

```
wget https://github.com/AMDResearch/omniperf/releases/download/v1.1.0-PR1/omniperf-v1.1.0-PR1.tar.gz

tar zxvf  omniperf-v1.1.0-PR1.tar.gz

cd omniperf-1.1.0-PR1/
python3 -m pip install -t ${INSTALL_DIR}/python-libs -r requirements.txt
mkdir build
cd build
export PYTHONPATH=$INSTALL_DIR/python-libs:$PYTHONPATH
cmake -DCMAKE_INSTALL_PREFIX=${INSTALL_DIR}/1.0.10 \
        -DPYTHON_DEPS=${INSTALL_DIR}/python-libs \
        -DMOD_INSTALL_PATH=${INSTALL_DIR}/modulefiles ..
make install
export PATH=$INSTALL_DIR/1.1.0_PR1/bin:$PATH
```

Dependencies

- ROCm (>=5.2), Python (>=3.7), CMake (>=3.19)

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:

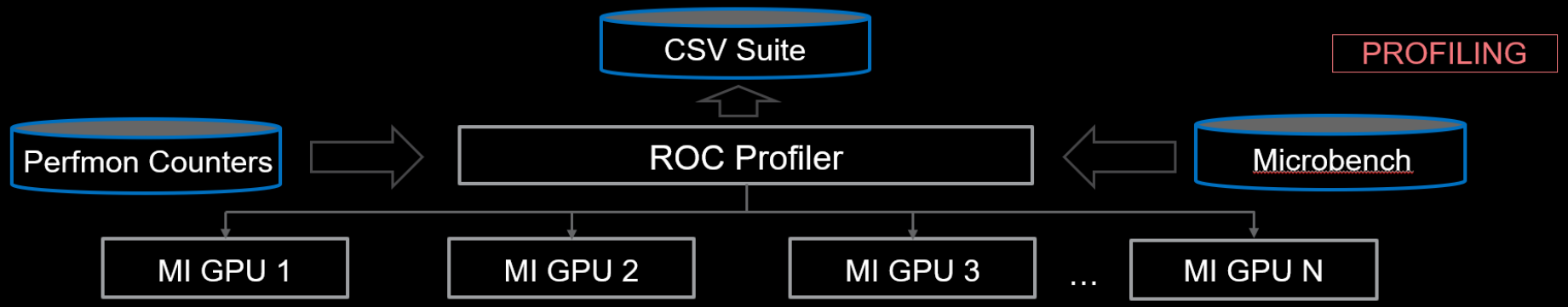
```
$ omniperf database <interaction type> [connection options]
```

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

Profile Mode

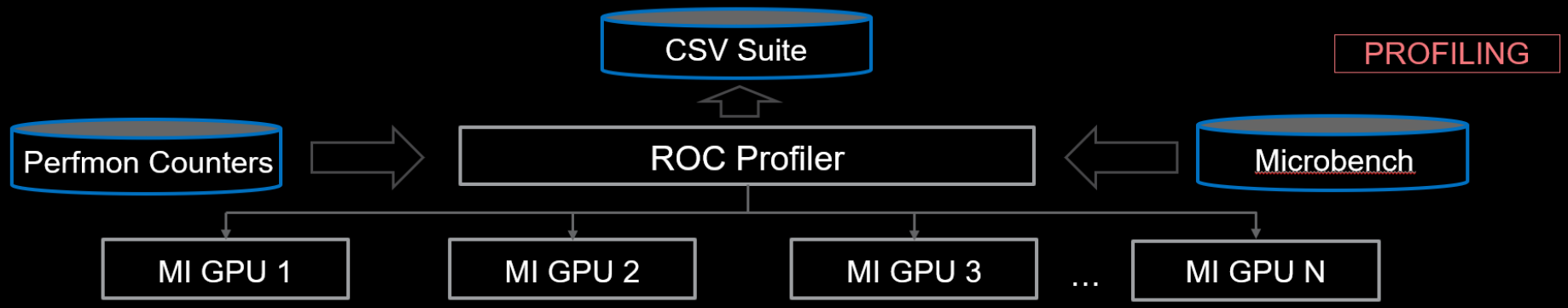


Features:

- 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.

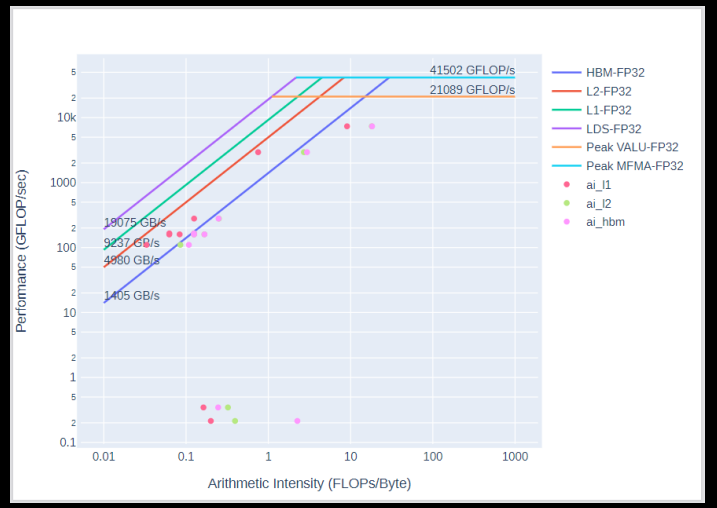
Profile Mode



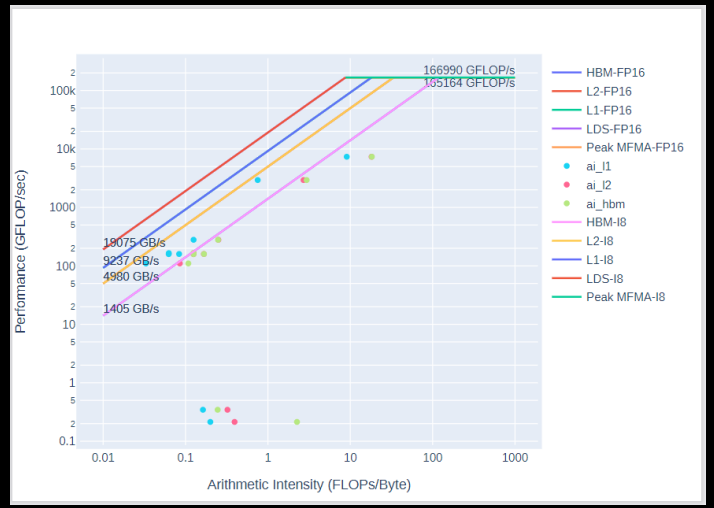
Features:

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

FP32/FP64

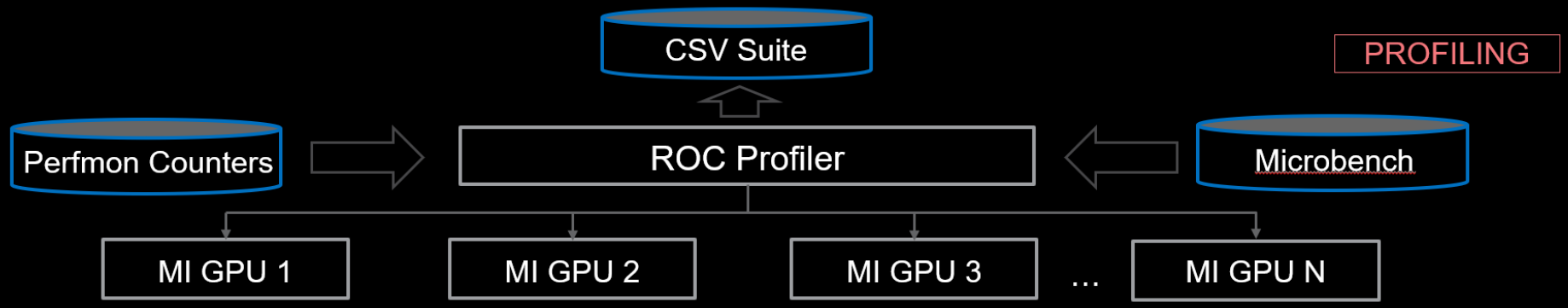


FP16/INT8



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

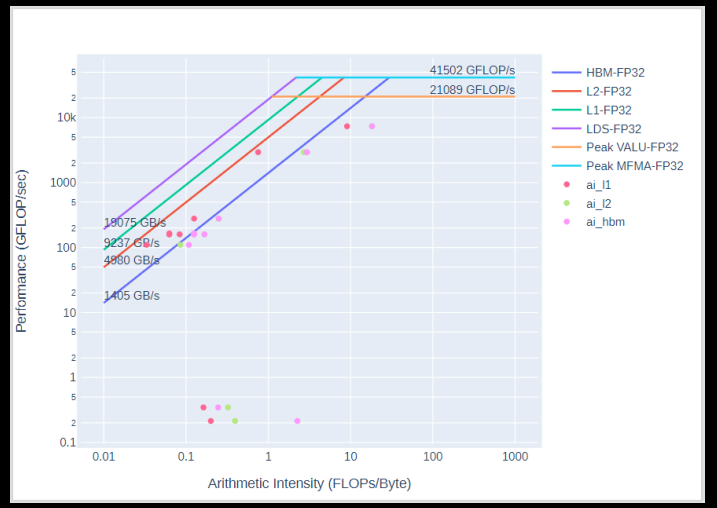
Profile Mode



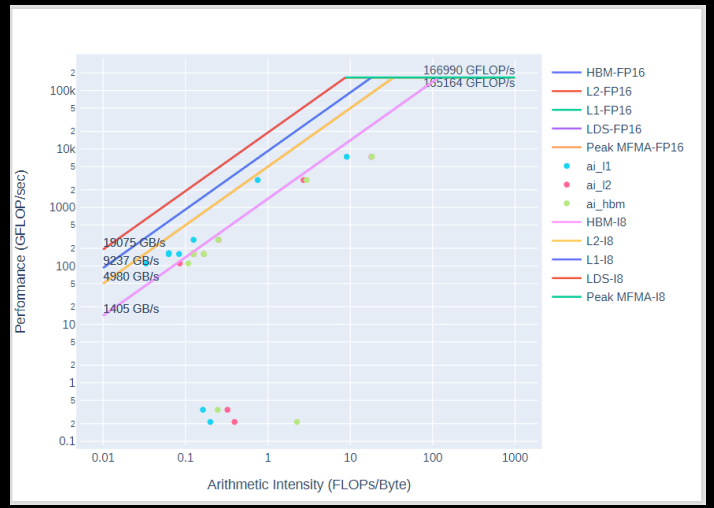
Features:

- 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

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
```

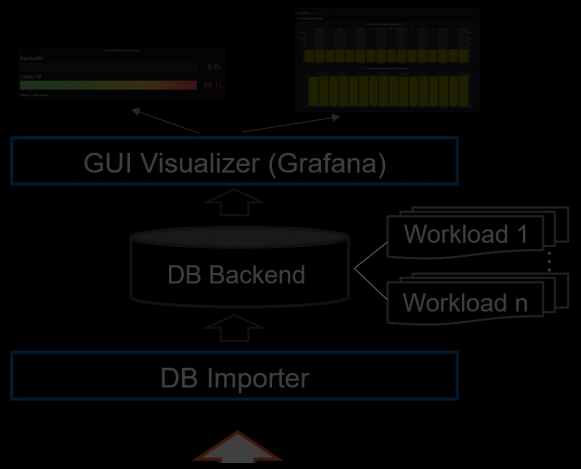
Profile with Omniperf:

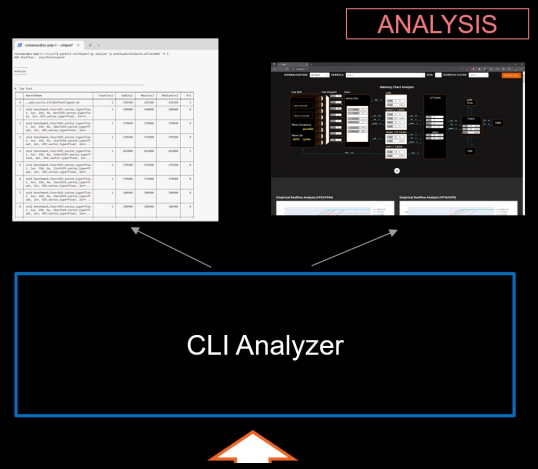
```
$ omniperf profile –n vcopy_all -- ./vcopy 1048576 256
…
-------------
Profile only
-------------

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`

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
```

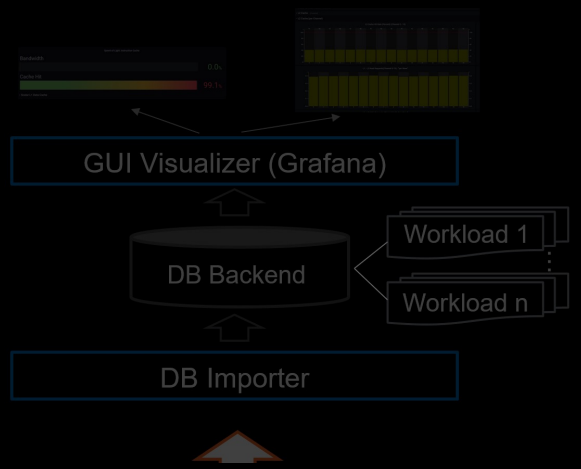
	KernelName
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vecto
2	void benchmark_func<double, 256, 8u, 512u>(double, double*) [clone .kd]
3	void benchmark_func<int, 256, 8u, 256u>(int, int*) [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]

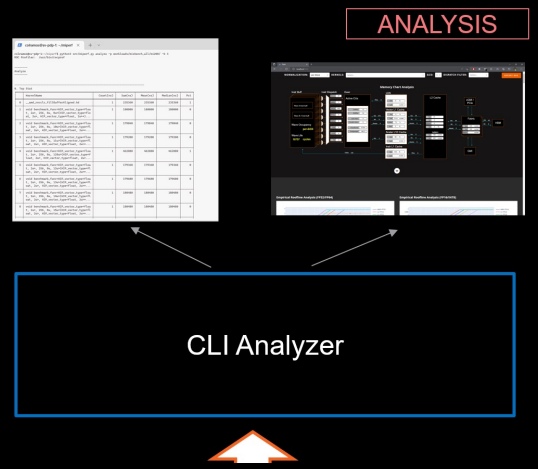
```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --list-metrics gfx90a
```

	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:

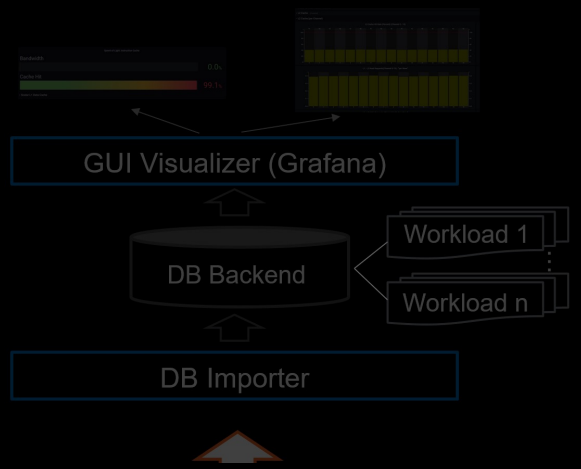
- List top kernels or view list of metrics
  `--list-kernels`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`

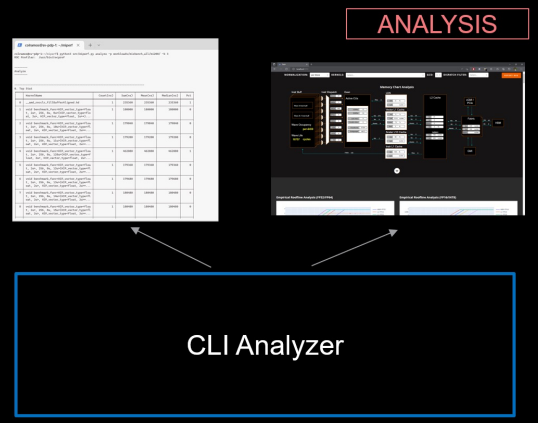
```
colramos@sv-pdp-2:~/GitHub/omniperf-pub$ ./src/omniperf analyze -p workloads/mix_all/mi200/ --kernel 0

--------
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
```

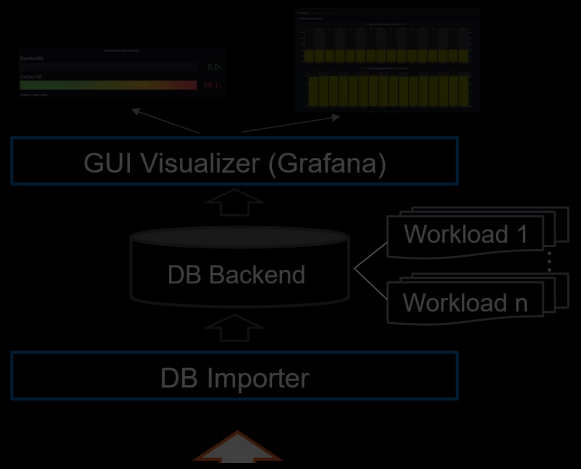
	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	void benchmark_func<int, 256, 8u, 512u>(int, int*) [clone .kd]	1	3353042.00	3353042.00	3353042.00	7.87
1	void benchmark_func<HIP_vector_type<float, 2u>, 256, 8u, 512u>(HIP_vector_type<float, 2u>, HIP_vector_type<float, 2u>...	1	1721239.00	1721239.00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

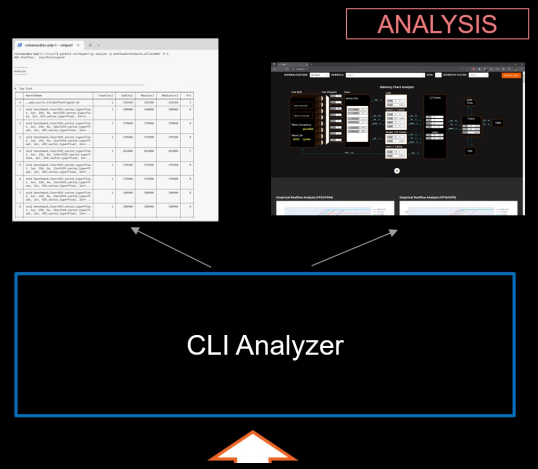
```
--------------------------------------------------------------------------------
5. Command Processor (CPC/CPF)
5.1 Command Processor Fetcher
```

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:

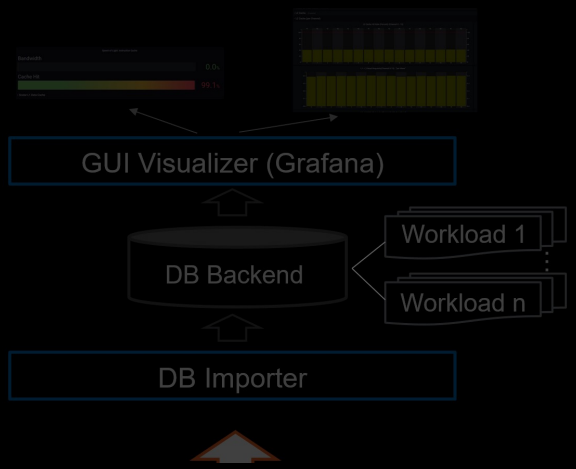
- List top kernels or view list of metrics
  `--list-kernels`, `--list-metrics`
- Filter available kernels, dispatches, gpu-ids
  `--kernel`, `--dispatch`, `--gpu-id`
- Filter by metric id(s)
  `--metric`
- Change normalization unit, time unit, or decimal
  `--normal-unit`, `--time-unit`, `--decimal`

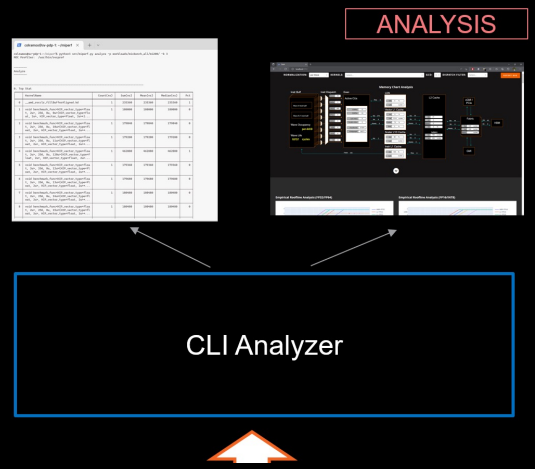
7.2 Wavefront Runtime Stats

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>
```

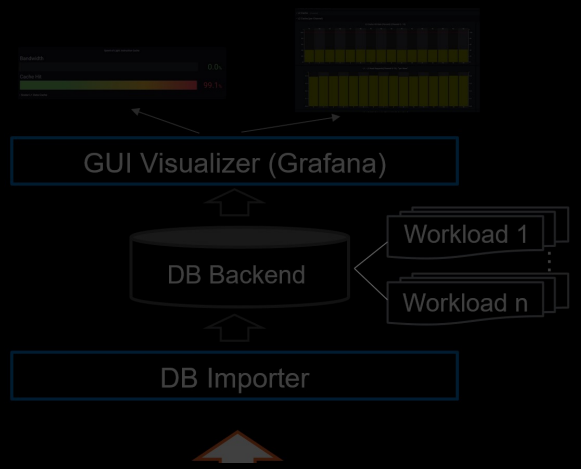
2. System Speed-of-Light

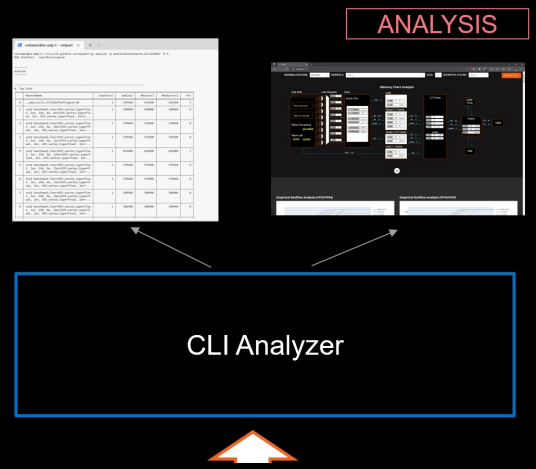
Index	Metric	Value	Value	Unit	Peak	Peak	PoP	PoP
2.1.0	VALU FLOPs	7492.7178288728755	0.0 (-100.0%)	GFlop	22630.4	22630.4 (0.0%)	33.10908260071795	0.0 (-100.0%)
2.1.1	VALU IOPs	2326.1937250093497	398.91 (-82.85%)	Giop	22630.4	22630.4 (0.0%)	10.279065880449968	1.76 (-82.85%)
2.1.2	MFMA FLOPs (BF16)	0.0	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

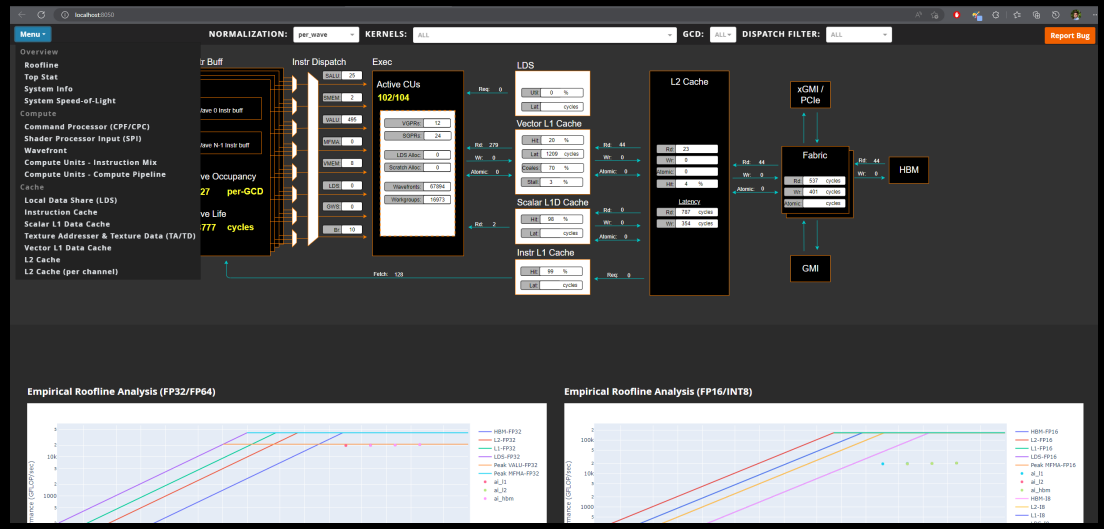
  `--path <workload1_path> --path <workload2_path>`

- 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

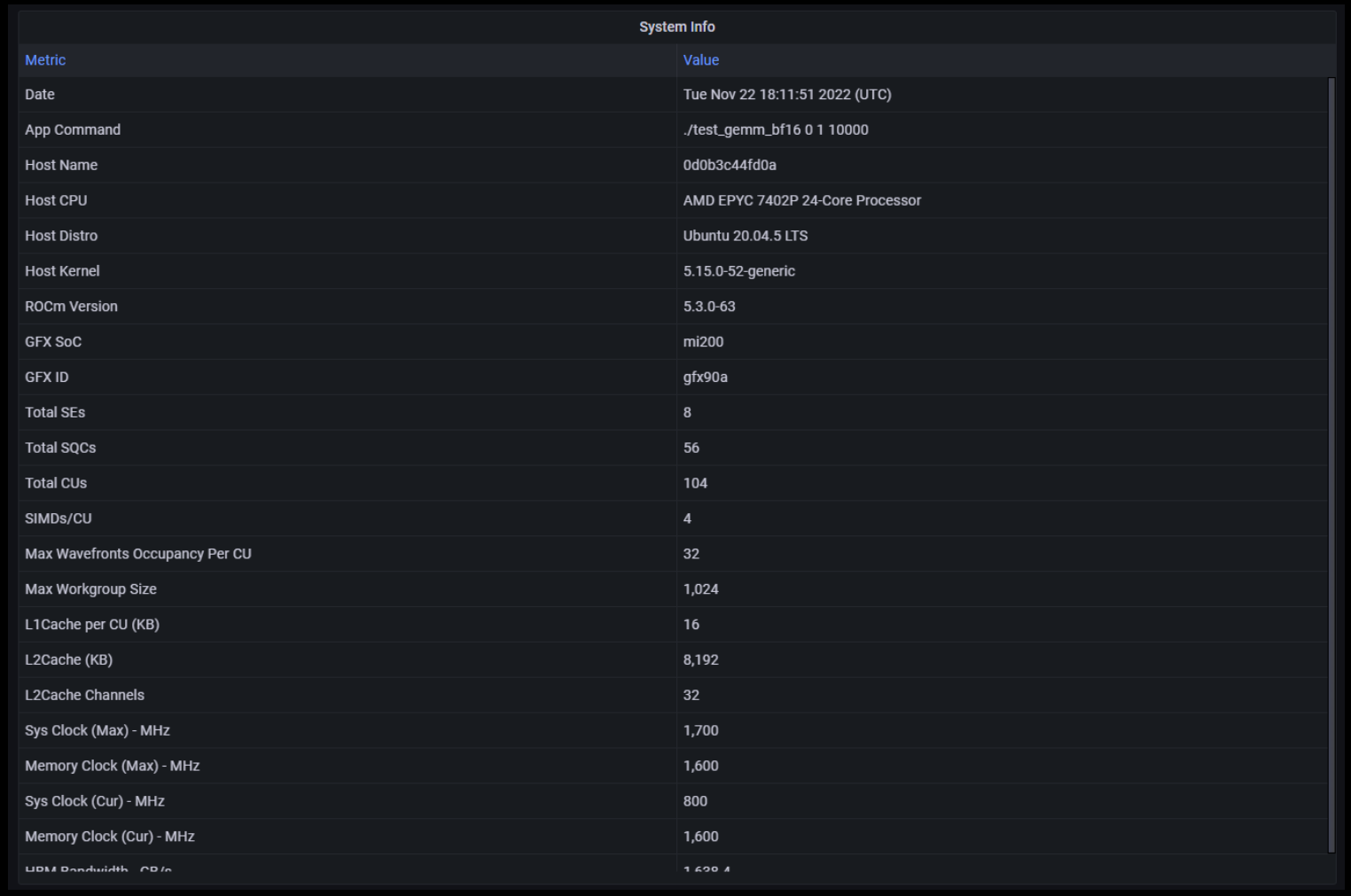


The above webpage is launched when the --gui option is used

Key Insights from Omniperf Analyzer

High level Metrics

- System Info



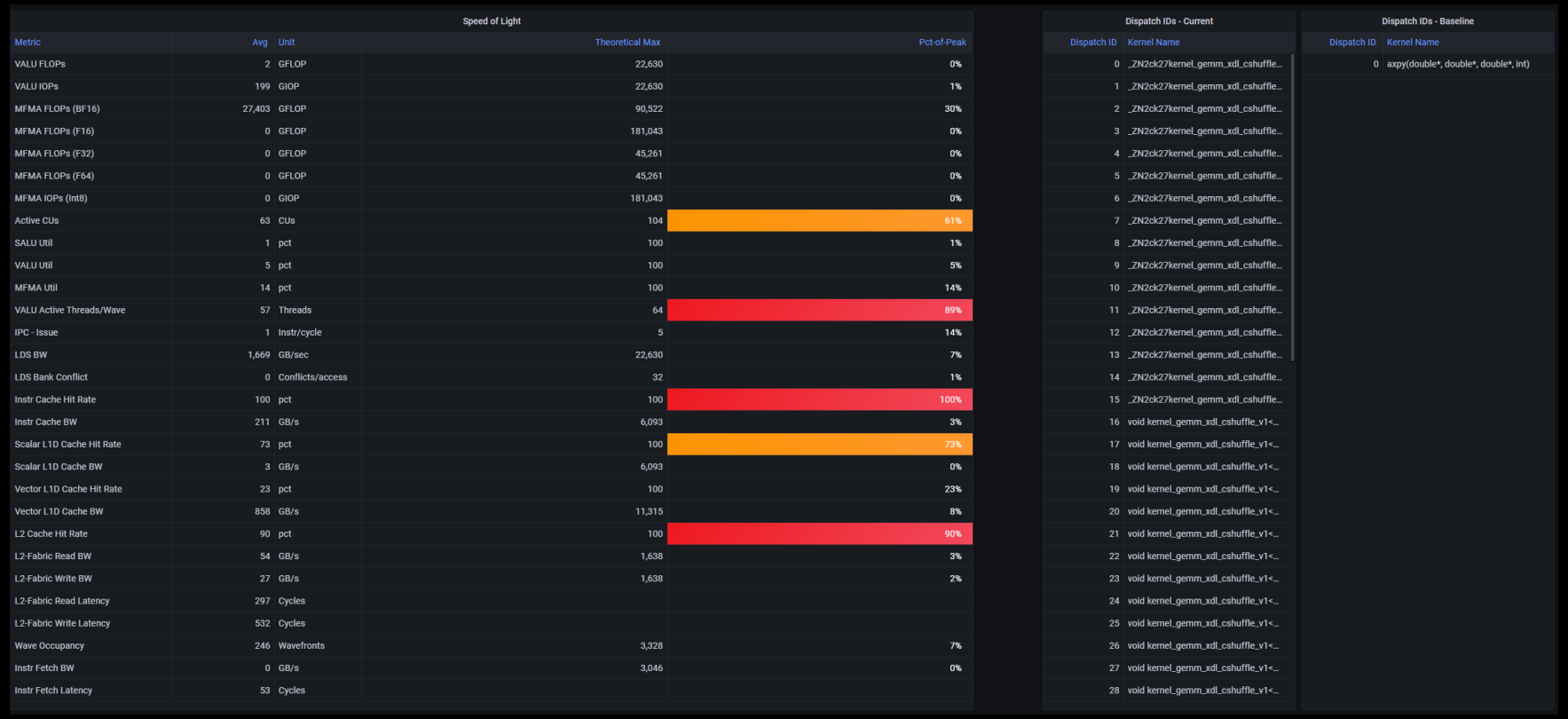
System Info

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
HBM Bandwidth - GB/s	1,638.4

Detailed system info for each app is collected by default

High level Metrics

- System Info
- System Speed-of-Light



Speed of Light

Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
VALU FLOPs	2	GFLOP	22,630	0%
VALU IOPs	199	GIOP	22,630	1%
MFMA FLOPs (BF16)	27,403	GFLOP	90,522	30%
MFMA FLOPs (F16)	0	GFLOP	181,043	0%
MFMA FLOPs (F32)	0	GFLOP	45,261	0%
MFMA FLOPs (F64)	0	GFLOP	45,261	0%
MFMA IOPs (Int8)	0	GIOP	181,043	0%
Active CUs	63	CUs	104	61%
SALU Util	1	pct	100	1%
VALU Util	5	pct	100	5%
MFMA Util	14	pct	100	14%
VALU Active Threads/Wave	57	Threads	64	89%
IPC - Issue	1	Instr/cycle	5	14%
LDS BW	1,669	GB/sec	22,630	7%
LDS Bank Conflict	0	Conflicts/access	32	1%
Instr Cache Hit Rate	100	pct	100	100%
Instr Cache BW	211	GB/s	6,093	3%
Scalar L1D Cache Hit Rate	73	pct	100	73%
Scalar L1D Cache BW	3	GB/s	6,093	0%
Vector L1D Cache Hit Rate	23	pct	100	23%
Vector L1D Cache BW	858	GB/s	11,315	8%
L2 Cache Hit Rate	90	pct	100	90%
L2-Fabric Read BW	54	GB/s	1,638	3%
L2-Fabric Write BW	27	GB/s	1,638	2%
L2-Fabric Read Latency	297	Cycles		
L2-Fabric Write Latency	532	Cycles		
Wave Occupancy	246	Wavefronts	3,328	7%
Instr Fetch BW	0	GB/s	3,046	0%
Instr Fetch Latency	53	Cycles		

Dispatch IDs - Current

Dispatch ID	Kernel Name
0	_ZN2ck27kernel_gemm_xdl_cshuffle...
1	_ZN2ck27kernel_gemm_xdl_cshuffle...
2	_ZN2ck27kernel_gemm_xdl_cshuffle...
3	_ZN2ck27kernel_gemm_xdl_cshuffle...
4	_ZN2ck27kernel_gemm_xdl_cshuffle...
5	_ZN2ck27kernel_gemm_xdl_cshuffle...
6	_ZN2ck27kernel_gemm_xdl_cshuffle...
7	_ZN2ck27kernel_gemm_xdl_cshuffle...
8	_ZN2ck27kernel_gemm_xdl_cshuffle...
9	_ZN2ck27kernel_gemm_xdl_cshuffle...
10	_ZN2ck27kernel_gemm_xdl_cshuffle...
11	_ZN2ck27kernel_gemm_xdl_cshuffle...
12	_ZN2ck27kernel_gemm_xdl_cshuffle...
13	_ZN2ck27kernel_gemm_xdl_cshuffle...
14	_ZN2ck27kernel_gemm_xdl_cshuffle...
15	_ZN2ck27kernel_gemm_xdl_cshuffle...
16	void kernel_gemm_xdl_cshuffle_v1<...
17	void kernel_gemm_xdl_cshuffle_v1<...
18	void kernel_gemm_xdl_cshuffle_v1<...
19	void kernel_gemm_xdl_cshuffle_v1<...
20	void kernel_gemm_xdl_cshuffle_v1<...
21	void kernel_gemm_xdl_cshuffle_v1<...
22	void kernel_gemm_xdl_cshuffle_v1<...
23	void kernel_gemm_xdl_cshuffle_v1<...
24	void kernel_gemm_xdl_cshuffle_v1<...
25	void kernel_gemm_xdl_cshuffle_v1<...
26	void kernel_gemm_xdl_cshuffle_v1<...
27	void kernel_gemm_xdl_cshuffle_v1<...
28	void kernel_gemm_xdl_cshuffle_v1<...

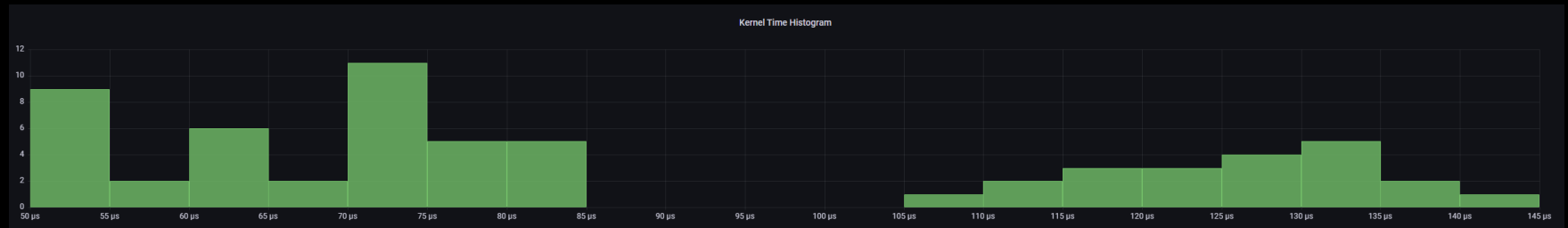
Dispatch IDs - Baseline

Dispatch ID	Kernel Name
0	axpy(double*, double*, double*, int)

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

High level Metrics

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



Top Kernels

Name	Calls	Performance	HBM BW	Total Duration	Avg Duration	AI (Vector L1D Cache	AI (L2 Cache)	AI (HBM)	Total FLOPs	VALU FLOPs	MFMA FLOPs (F16)	MFMA FLOPs (BF16)	MFMA FLOPs (F32)	MFMA FLOPs (F64)	LDS	Vector L1D Cache	L2 Cache
void kernel_gemm_xdl...	6	22.4 TFLOPS	65.9 GB/s	619 µs	103 µs	39.386	56.891	340.579	2,147,581,952	98,304	0	2,147,483,648	0	0	98.6 MB	54,525,952	37.7 MB
_ZN2ck27kernel_gem...	1	14.9 TFLOPS	43.9 GB/s	144 µs	144 µs	39.386	73.145	340.364	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	54,525,952	29.4 MB
_ZN2ck27kernel_gem...	1	15.7 TFLOPS	46.1 GB/s	137 µs	137 µs	46.547	73.145	340.364	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	46,137,344	29.4 MB
_ZN2ck27kernel_gem...	1	16.0 TFLOPS	47.0 GB/s	134 µs	134 µs	39.386	73.145	340.378	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	54,525,952	29.4 MB
void kernel_gemm_xdl...	2	32.3 TFLOPS	94.8 GB/s	134 µs	67.1 µs	39.388	39.388	340.922	2,147,680,256	196,608	0	2,147,483,648	0	0	141 MB	54,525,952	54.5 MB

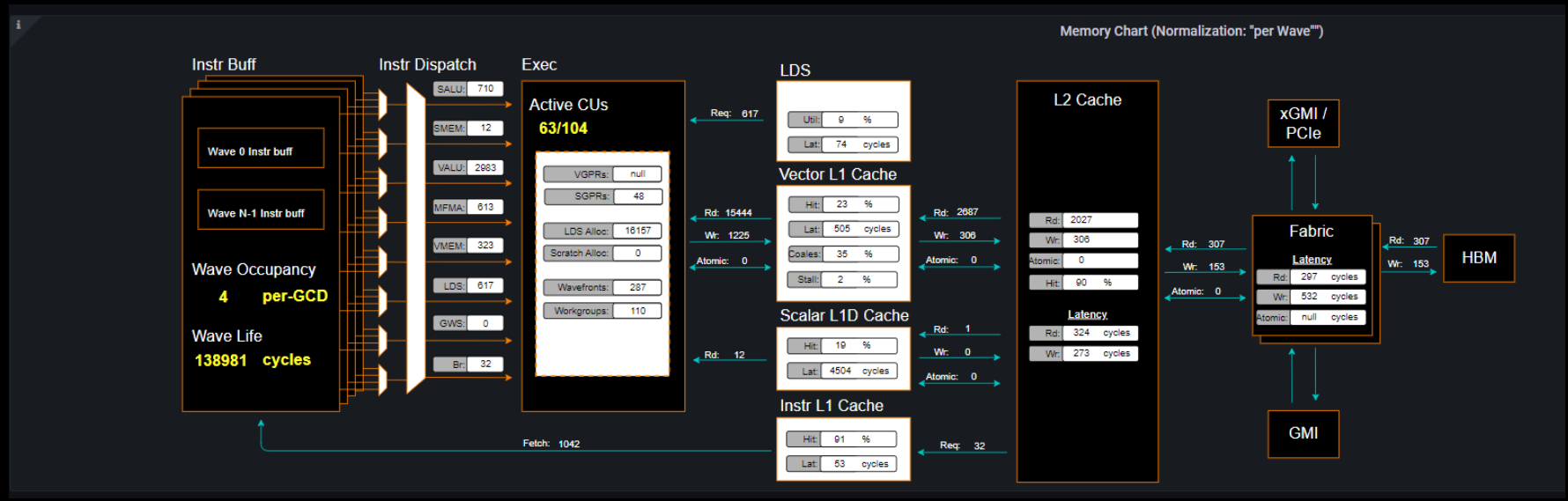
Top Dispatches

Dispatch	Calls	Performance	HBM BW	Total Duration	Avg Duration	AI (Vector L1D Cache	AI (L2 Cache)	AI (HBM)	Total FLOPs	VALU FLOPs	MFMA FLOPs (F16)	MFMA FLOPs (BF16)	MFMA FLOPs (F32)	MFMA FLOPs (F64)	LDS	Vector L1D Cache	L2 Cache	HE
31	1	14.9 TFLOPS	43.9 GB/s	144 µs	144 µs	39.386	73.145	340.364	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	54,525,952	29.4 MB	6.31 M
33	1	15.4 TFLOPS	45.3 GB/s	139 µs	139 µs	30.119	56.891	340.336	2,147,549,184	65,536	0	2,147,483,648	0	0	90.2 MB	71,303,168	37.7 MB	6.31 M
32	1	15.7 TFLOPS	46.1 GB/s	137 µs	137 µs	46.547	73.145	340.364	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	46,137,344	29.4 MB	6.31 M
34	1	16.0 TFLOPS	46.9 GB/s	135 µs	135 µs	30.119	56.891	340.35	2,147,549,184	65,536	0	2,147,483,648	0	0	90.2 MB	71,303,168	37.7 MB	6.31 M
29	1	16.0 TFLOPS	47.0 GB/s	134 µs	134 µs	39.386	73.145	340.378	2,147,549,184	65,536	0	2,147,483,648	0	0	81.8 MB	54,525,952	29.4 MB	6.31 M

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

High level Metrics

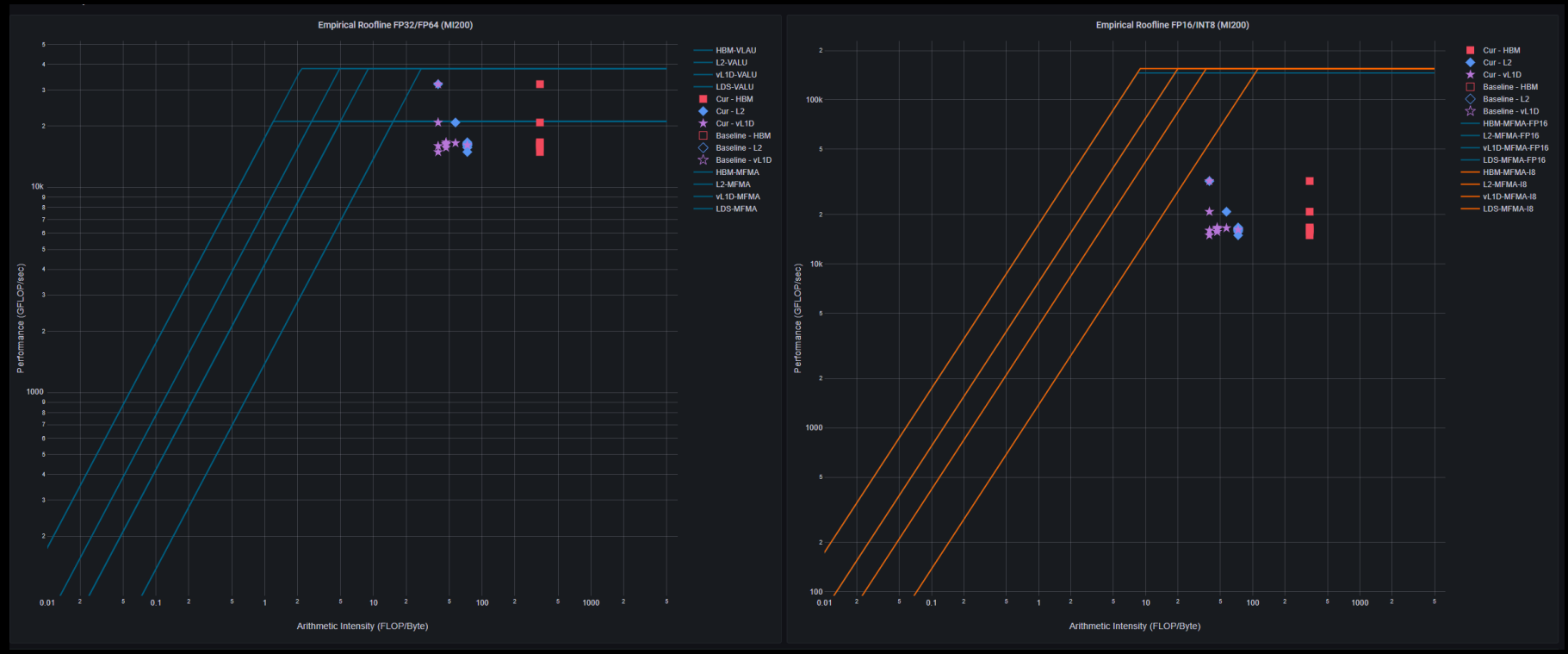
- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis



Illustrate data movement and performance on key components of target architecture

High level Metrics

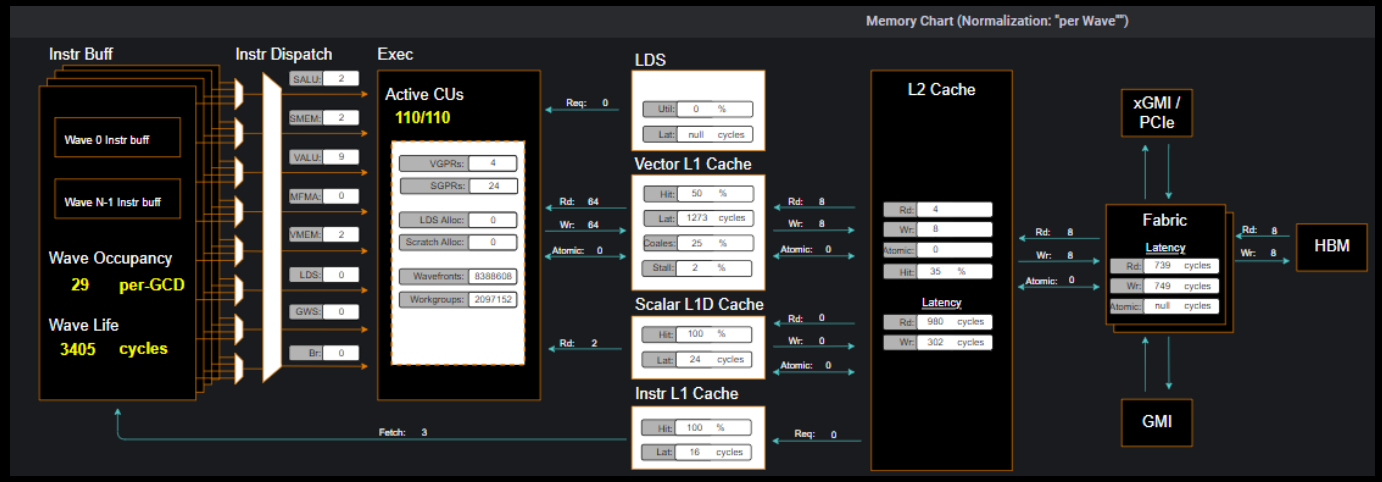
- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis
- Roofline Analysis



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

Initial assessment with kernel statistics

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

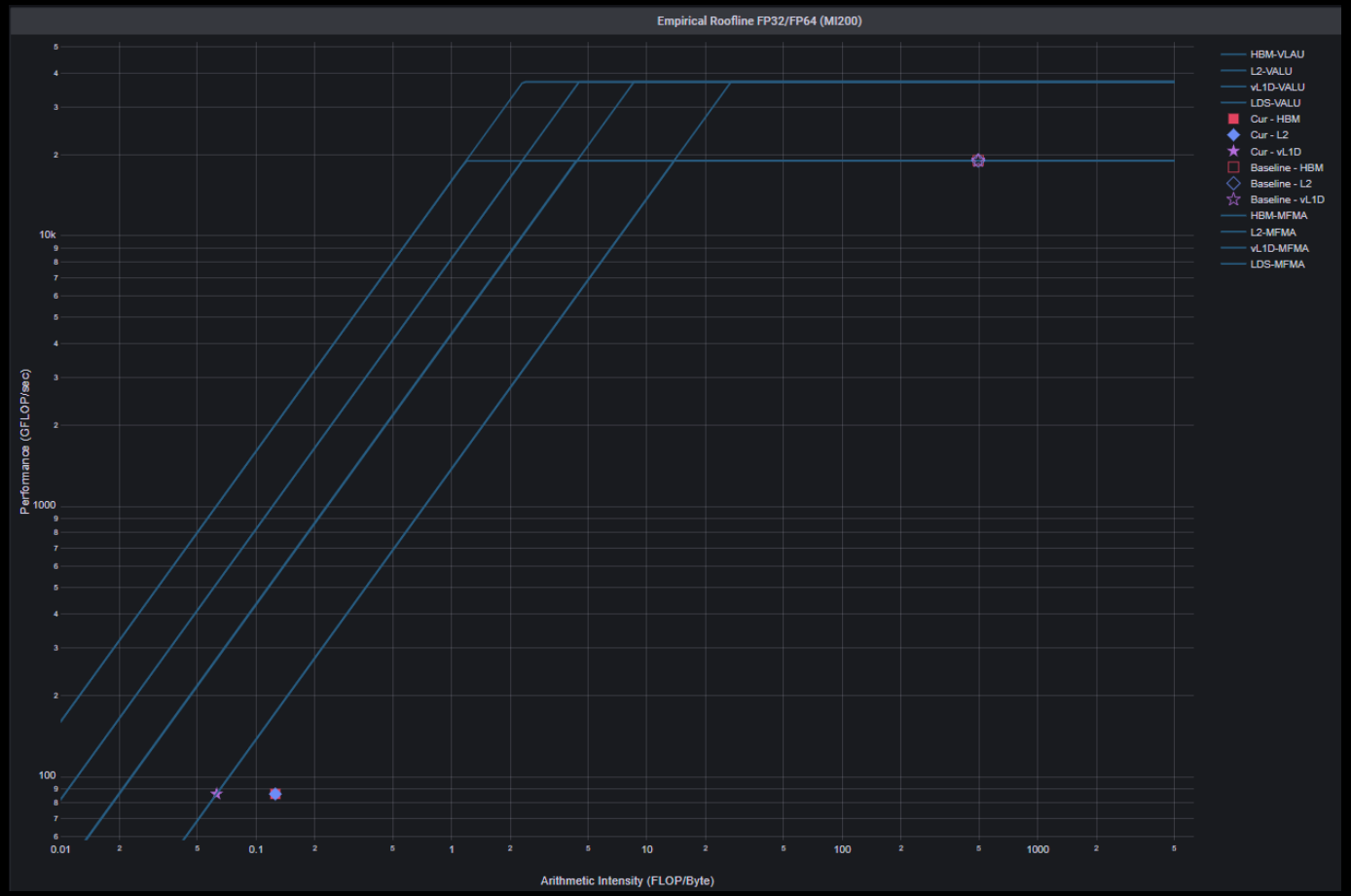


Speed of Light

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

Roofline: the first-step characterization of workload performance

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



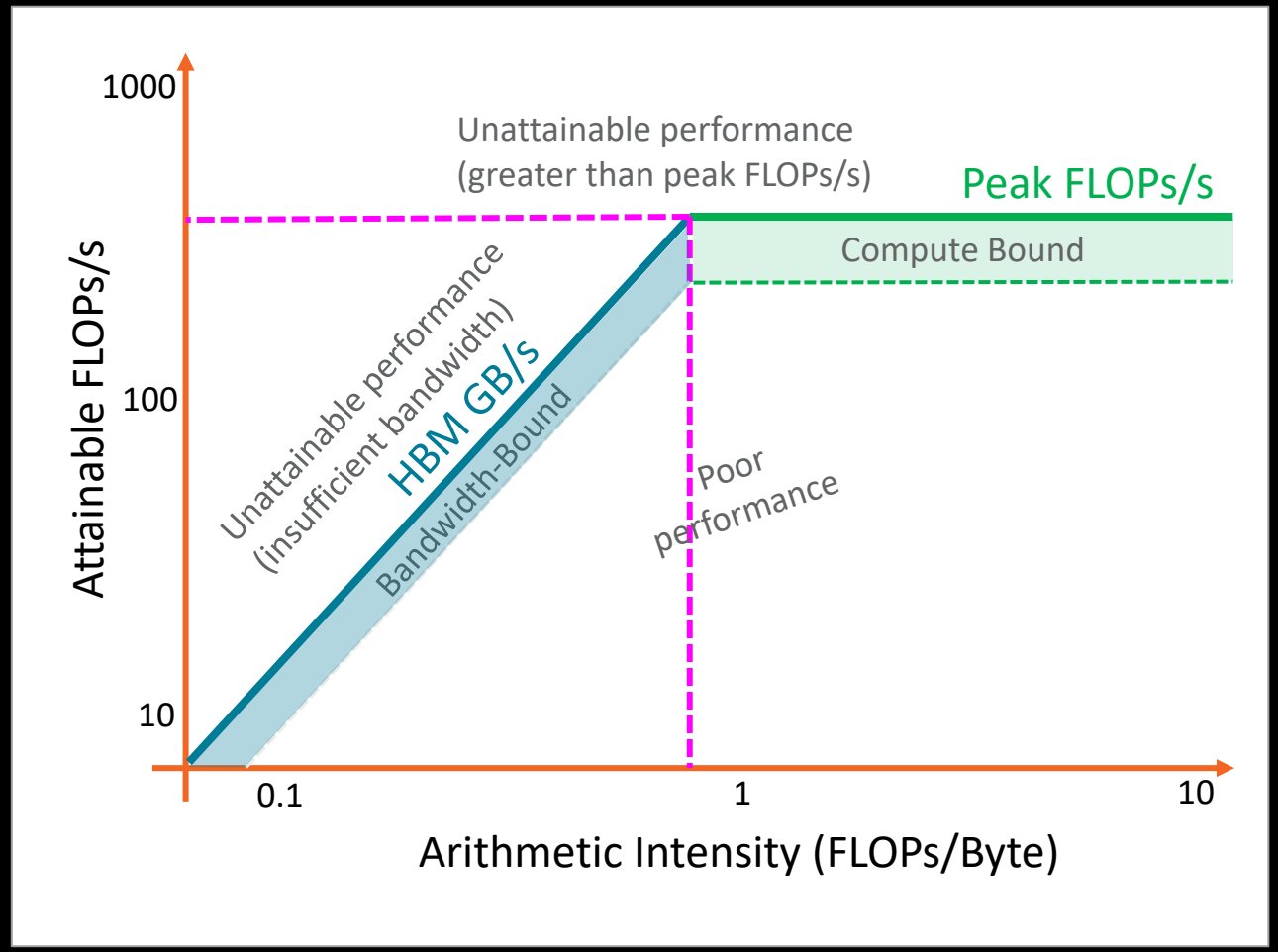
Top Kernels

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

Background - What is a roofline?

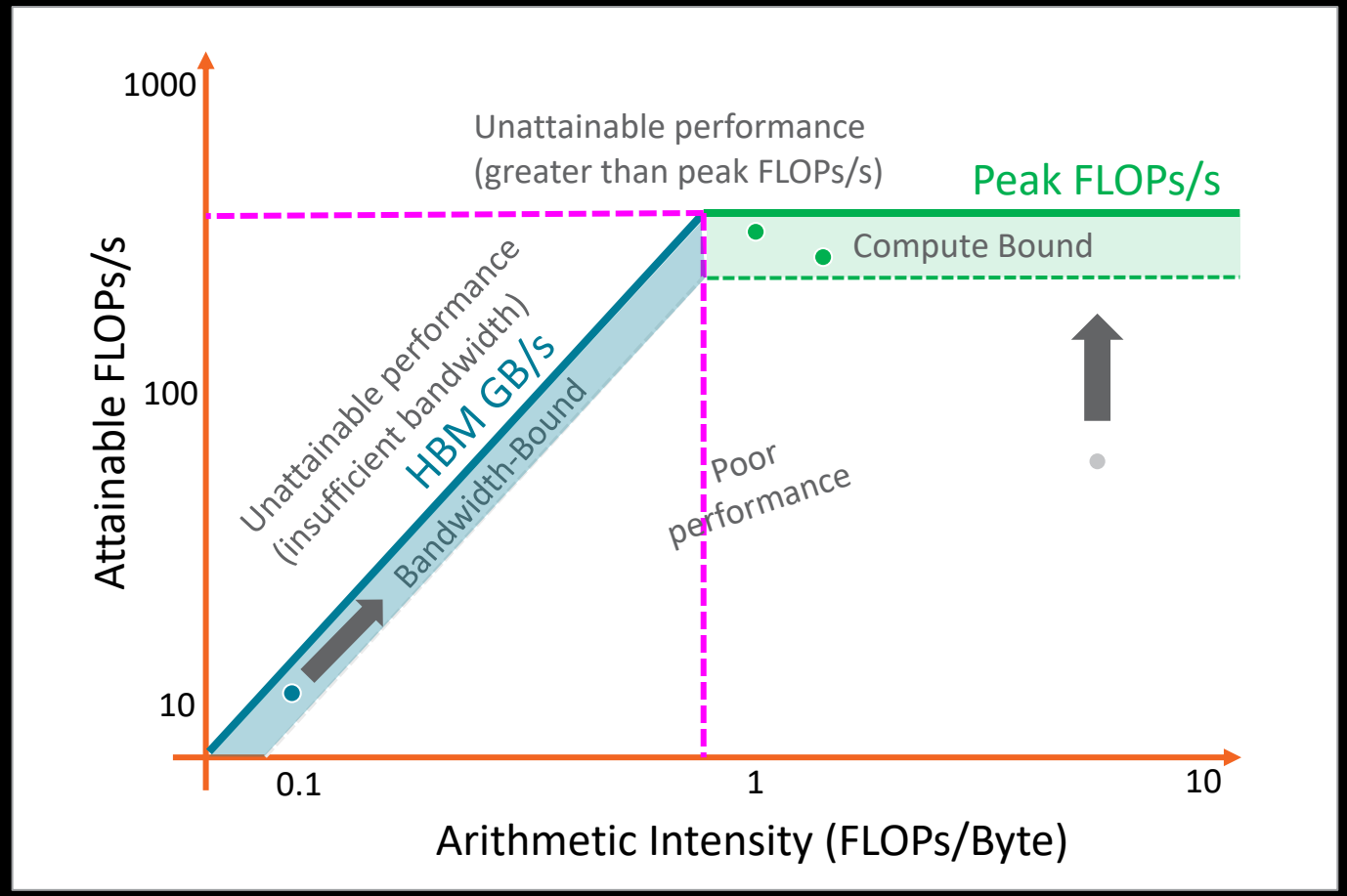
Background – What is roofline?

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

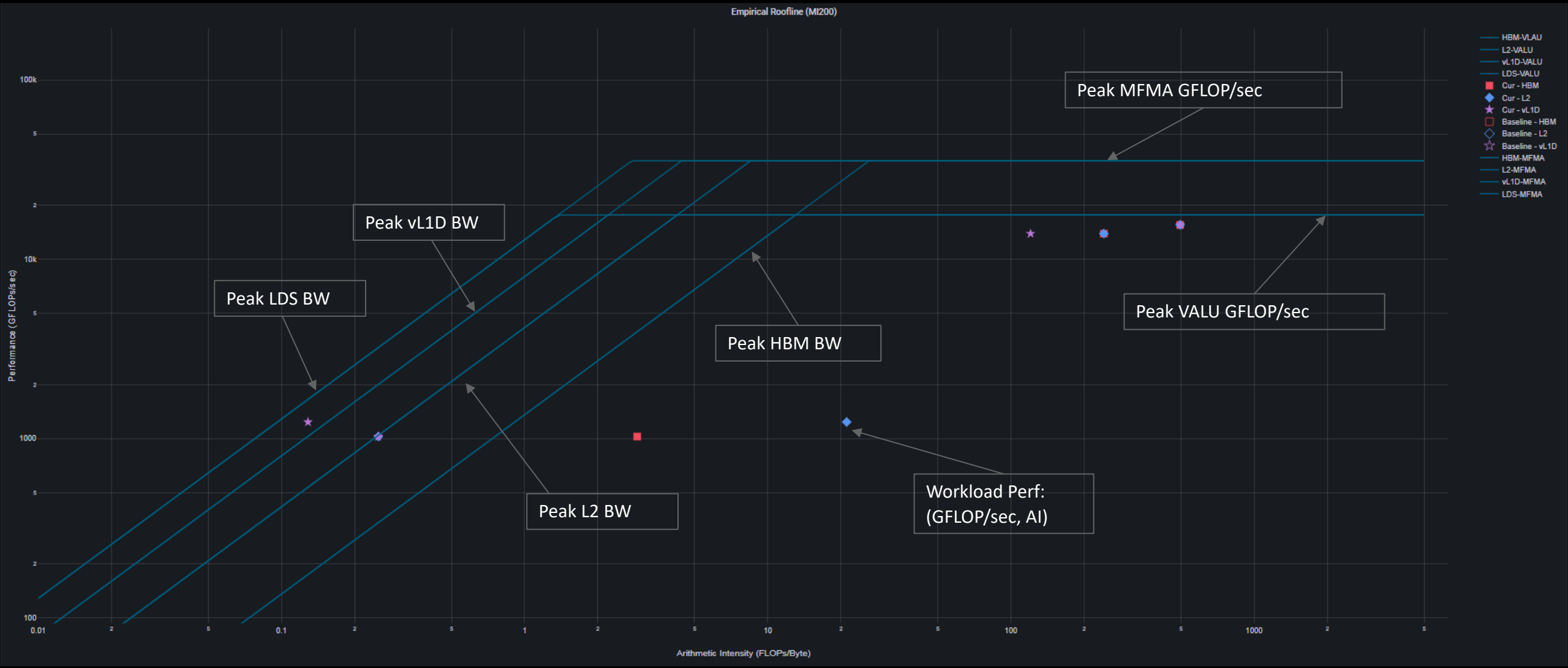


Background – What is roofline?

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

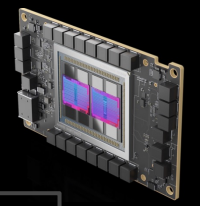


Empirical Hierarchical Roofline on MI200 - Overview



Roofline Calculations on AMD Instinct™ MI200 GPUs

Overview - AMD Instinct™ MI200 Architecture



Graphics Compute Die (GCD)

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

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

... ...

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

L2 Cache (L2)

(Peer GCD)

Data Fabric

Remote Socket (CPU, GPU)

Memory Controller

(GCD1)

(GCD2)

HBM Memory

HBM Memory

Empirical Hierarchical Roofline on MI200 – Roofline Benchmarking

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

```
10:57:35 amd@node-bp126-014a utils ±|master ✗|→ ./roofline
Total detected GPU devices: 2
GPU Device 0: Profiling...
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
HBM BW, GPU ID: 0, workgroupSize:256, workgroups:2097152, experiments:100, Total Bytes=8589934592, Duration=6.2 ms, Mean=1382.7 GB/sec, stdev=2.6 GB/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
L2 BW, GPU ID: 0, workgroupSize:256, workgroups:8192, experiments:100, Total Bytes=687194767360, Duration=157.3 ms, Mean=4321.3 GB/sec, stdev=59.1 GB/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
L1 BW, GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total Bytes=26843545600, Duration=3.3 ms, Mean=8262.6 GB/sec, stdev=5.9 GB/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
LDS BW, GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total Bytes=33554432000, Duration=1.8 ms, Mean=18780.4 GB/sec, stdev=33.0 GB/s
nSize:134217728, 268435456000
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak FLOPs (FP32), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=274877906944, Duration=14.482 ms, Mean=18977.7 GFLOPs/sec, stdev=3.6 GFLOPs/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak FLOPs (FP64), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=137438953472, Duration=7.5 ms, Mean=18336.156250.1 GFLOPs/sec, stdev=5.0 GFLOPs/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak MFMA FLOPs (BF16), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=2147483648000, Duration=14.0 ms, Mean=153763.7 GFLOPs/sec, stdev=61.0 GFLOPs/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak MFMA FLOPs (F16), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=2147483648000, Duration=14.5 ms, Mean=147890.9 GFLOPs/sec, stdev=32.2 GFLOPs/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak MFMA FLOPs (F32), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=536870912000, Duration=14.4 ms, Mean=37200.4 GFLOPs/sec, stdev=9.3 GFLOPs/s
 99% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
Peak MFMA FLOPs (F64), GPU ID: 0, workgroupSize:256, workgroups:16384, experiments:100, Total FLOPS=268435456000, Duration=7.3 ms, Mean=36978.4 GFLOPs/sec, stdev=10.0 GFLOPs/s
```

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

Empirical Hierarchical Roofline on MI200 - Arithmetic

$$\begin{aligned}\text{Total\_FLOP} = \quad & 64 * (\text{SQ\_INSTS\_VALU\_ADD\_F16} + \text{SQ\_INSTS\_VALU\_MUL\_F16} + \text{SQ\_INSTS\_VALU\_TRANS\_F16} + 2 * \text{SQ\_INSTS\_VALU\_FMA\_F16}) \\ & + 64 * (\text{SQ\_INSTS\_VALU\_ADD\_F32} + \text{SQ\_INSTS\_VALU\_MUL\_F32} + \text{SQ\_INSTS\_VALU\_TRANS\_F32} + 2 * \text{SQ\_INSTS\_VALU\_FMA\_F32}) \\ & + 64 * (\text{SQ\_INSTS\_VALU\_ADD\_F64} + \text{SQ\_INSTS\_VALU\_MUL\_F64} + \text{SQ\_INSTS\_VALU\_TRANS\_F64} + 2 * \text{SQ\_INSTS\_VALU\_FMA\_F64}) \\ & + 512 * \text{SQ\_INSTS\_VALU\_MFMA\_MOPS\_F16} \\ & + 512 * \text{SQ\_INSTS\_VALU\_MFMA\_MOPS\_BF16} \\ & + 512 * \text{SQ\_INSTS\_VALU\_MFMA\_MOPS\_F32} \\ & + 512 * \text{SQ\_INSTS\_VALU\_MFMA\_MOPS\_F64}\end{aligned}$$

$$\text{Total\_IOP} = 64 * (\text{SQ\_INSTS\_VALU\_INT32} + \text{SQ\_INSTS\_VALU\_INT64})$$

$$LDS_{BW} = 32 * 4 * (\text{SQ\_LDS\_IDX\_ACTIVE} - \text{SQ\_LDS\_BANK\_CONFLICT})$$

$$vL1D_{BW} = 64 * \text{TCP\_TOTAL\_CACHE\_ACCESSES\_sum}$$

$$\begin{aligned}L2_{BW} = \quad & 64 * \text{TCP\_TCC\_READ\_REQ\_sum} \\ & + 64 * \text{TCP\_TCC\_WRITE\_REQ\_sum} \\ & + 64 * (\text{TCP\_TCC\_ATOMIC\_WITH\_RET\_REQ\_sum} + \\ & \text{TCP\_TCC\_ATOMIC\_WITHOUT\_RET\_REQ\_sum})\end{aligned}$$

$$\begin{aligned}HBM_{BW} = \quad & 32 * \text{TCC\_EA\_RDREQ\_32B\_sum} + 64 * (\text{TCC\_EA\_RDREQ\_sum} - \\ & \text{TCC\_EA\_RDREQ\_32B\_sum}) \\ & + 32 * (\text{TCC\_EA\_WRREQ\_sum} - \text{TCC\_EA\_WRREQ\_64B\_sum}) + 64 * \\ & \text{TCC\_EA\_WRREQ\_64B\_sum}\end{aligned}$$

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

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

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

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

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

Empirical Hierarchical Roofline on MI200 - Manual Rocprof

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

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

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

```
## roof-counters.txt

# FP32 FLOPs
pmc: SQ_INSTS_VALU_ADD_F32 SQ_INSTS_VALU_MUL_F32 SQ_INSTS_VALU_FMA_F32 SQ_INSTS_VALU_TRANS_F32

# HBM Bandwidth
pmc: TCC_EA_RDREQ_sum TCC_EA_RDREQ_32B_sum TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_sum

# LDS Bandwidth
pmc: SQ_LDS_IDX_ACTIVE SQ_LDS_BANK_CONFLICT

# L2 Bandwidth
pmc: TCP_TCC_READ_REQ_sum TCP_TCC_WRITE_REQ_sum TCP_TCC_ATOMIC_WITH_RET_REQ_sum
TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum

# vL1D Bandwidth
pmc: TCP_TOTAL_CACHE_ACCESSES_sum
```

Low level Metrics

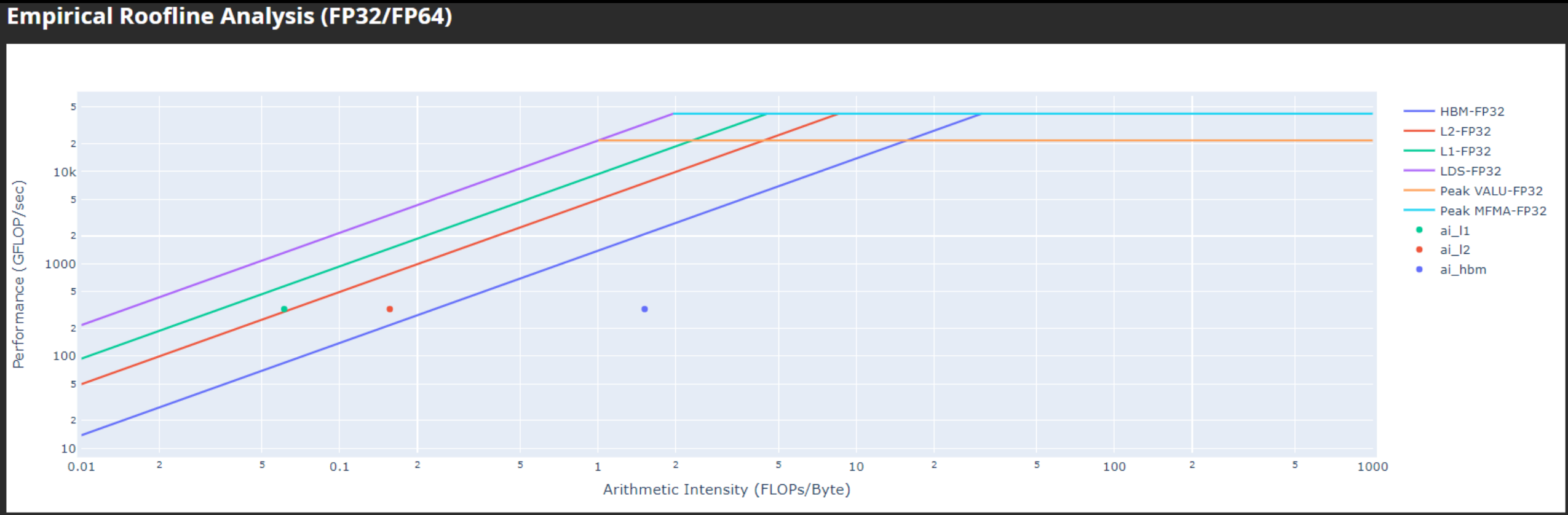
Section Title	Comments
Command Processor (CPC/CPF)	Packet processor data
Shader Processor Input (SPI)	Connecting packet processor and CUs
Wavefront Stats	Kernel launch stats
Compute Unit – Instruction Mix	Breakdown of instructions issued
Compute Unit – Compute Pipeline	
Texture Addressor & Texture Data (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)	

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



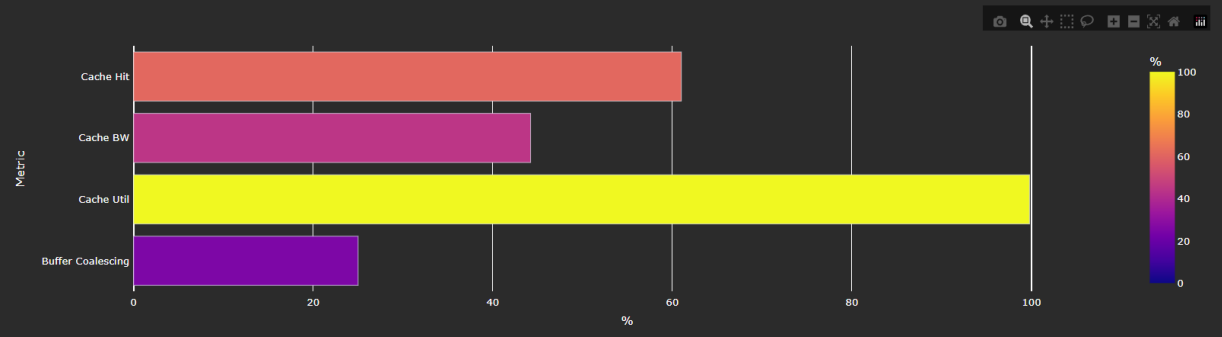
- Performance: almost 330 GFLOPs

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

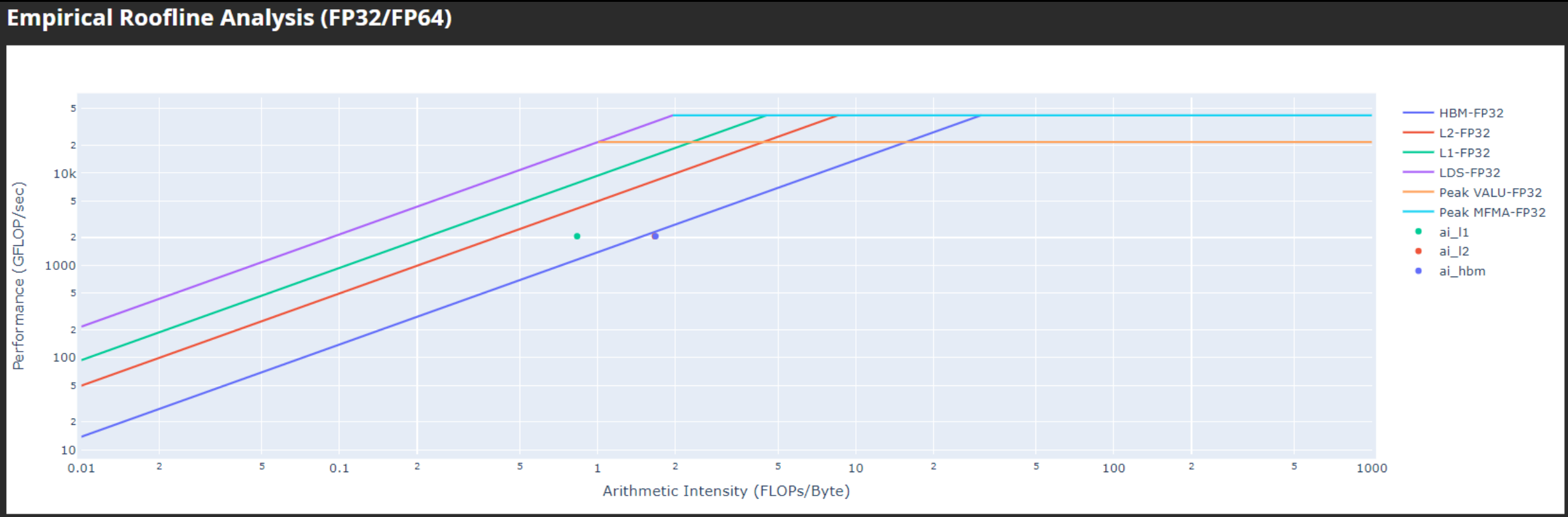
16.3 L1D Cache Accesses

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



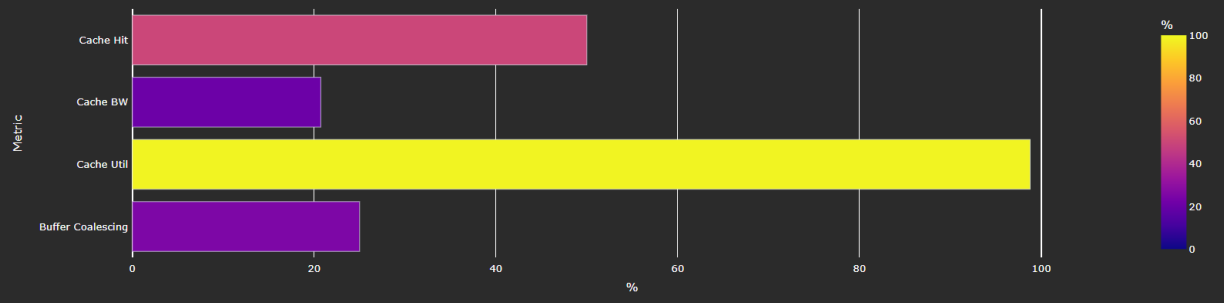
- Performance: almost 2 TFLOPs

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

Other Guided Exercises

https://github.com/olcf/hip-training-series/tree/master/Lecture5

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

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

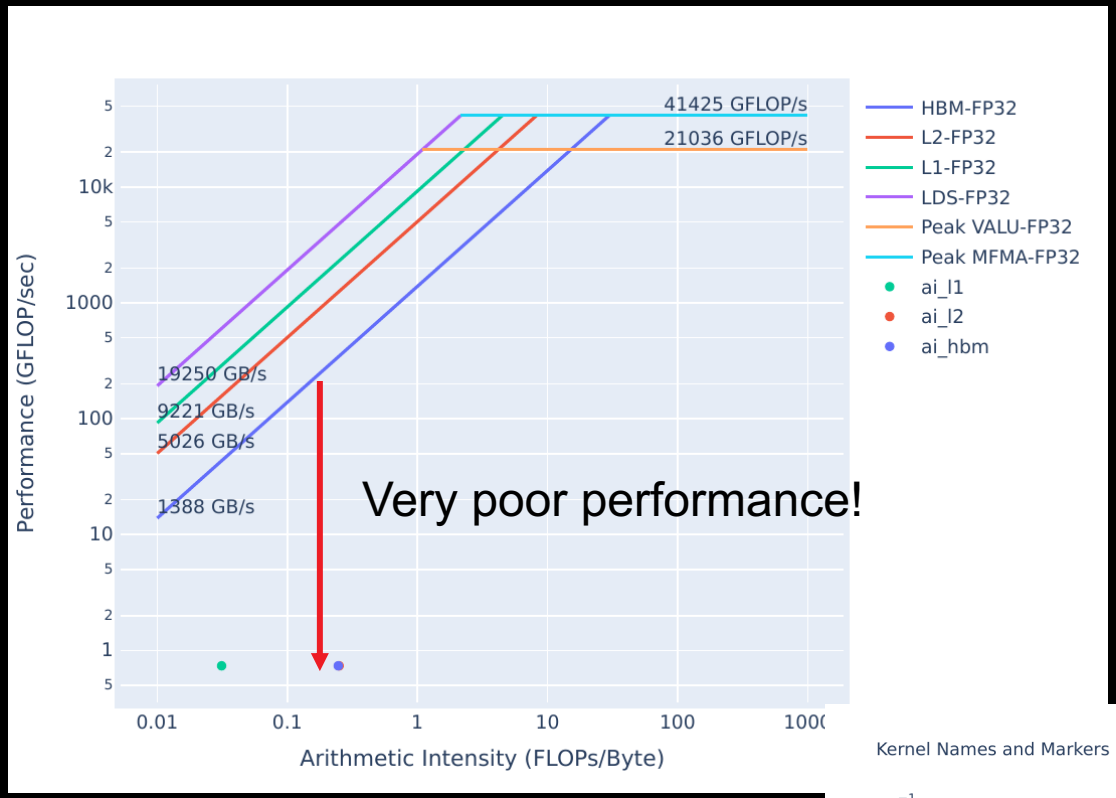
- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `omniperf profile –n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/mi200` folder.
    - `--roof-only` generates PDF roofline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

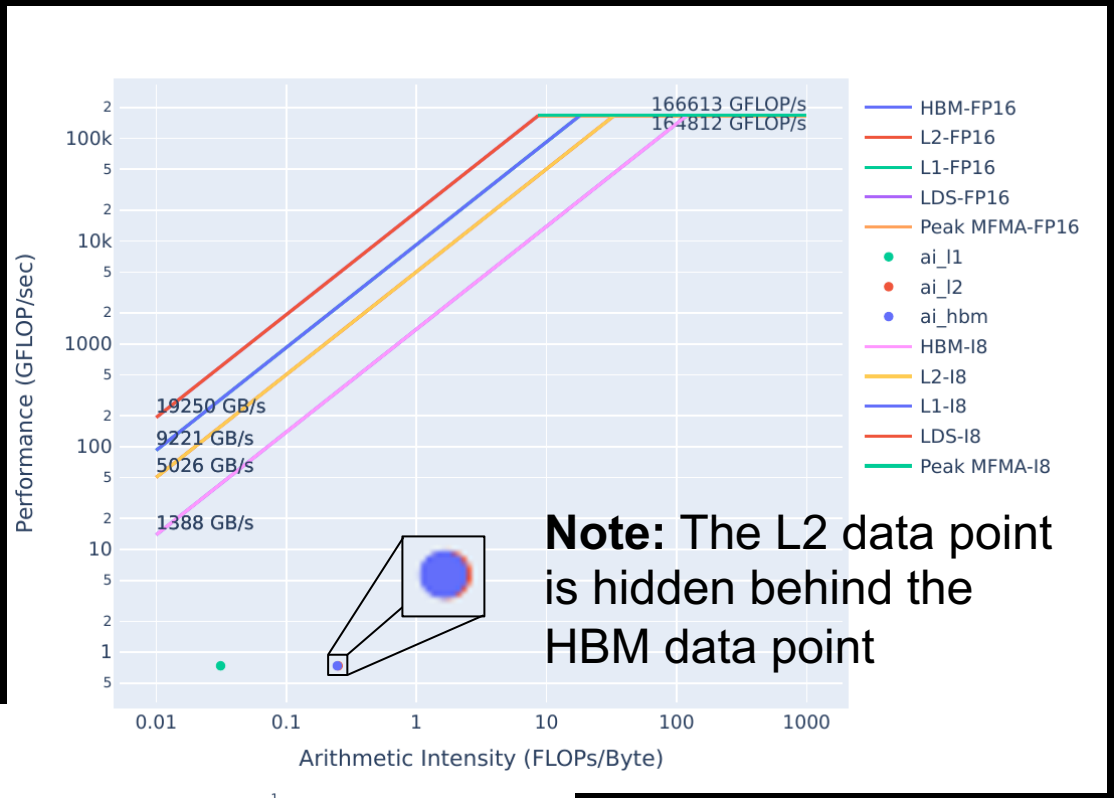
- Rooflines are a useful tool in determining which kernels are good optimization targets
  - They are only one perspective of performance: runtime of the kernel cannot be inferred from the roofline

- 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

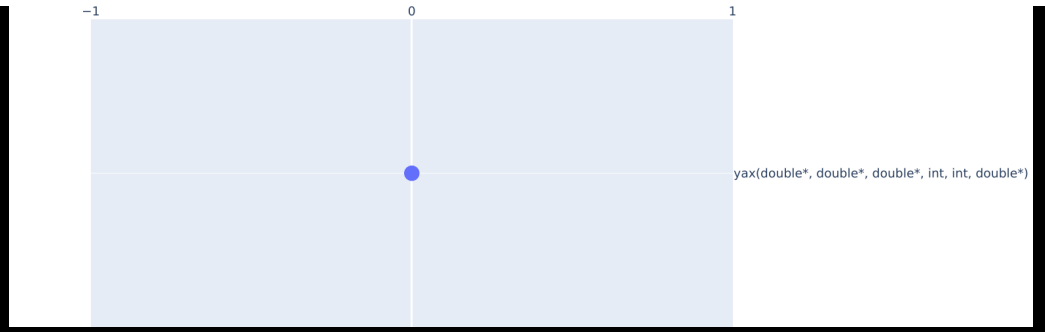
- Complete sets of Roofline plots and commands can be found in the READMEs for each exercise

Exercise 1: Problem Roofline Plots





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

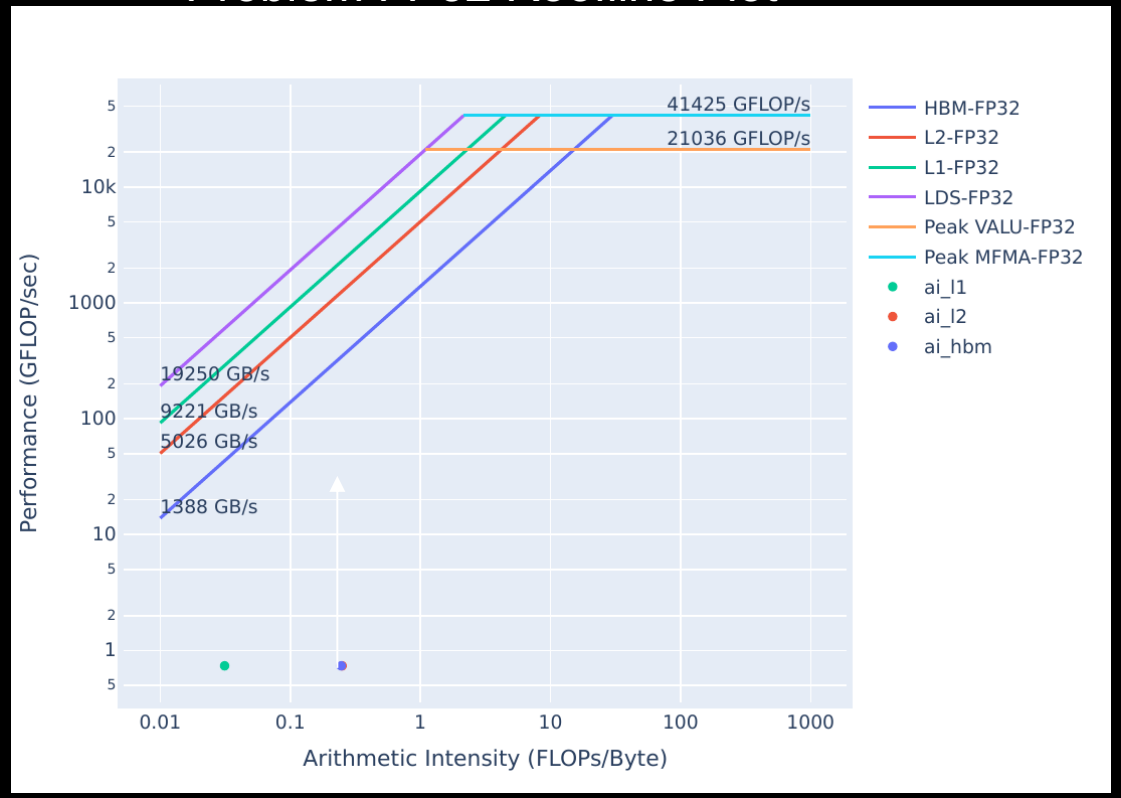
Increased launched wavefronts, which increases Grid Size

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

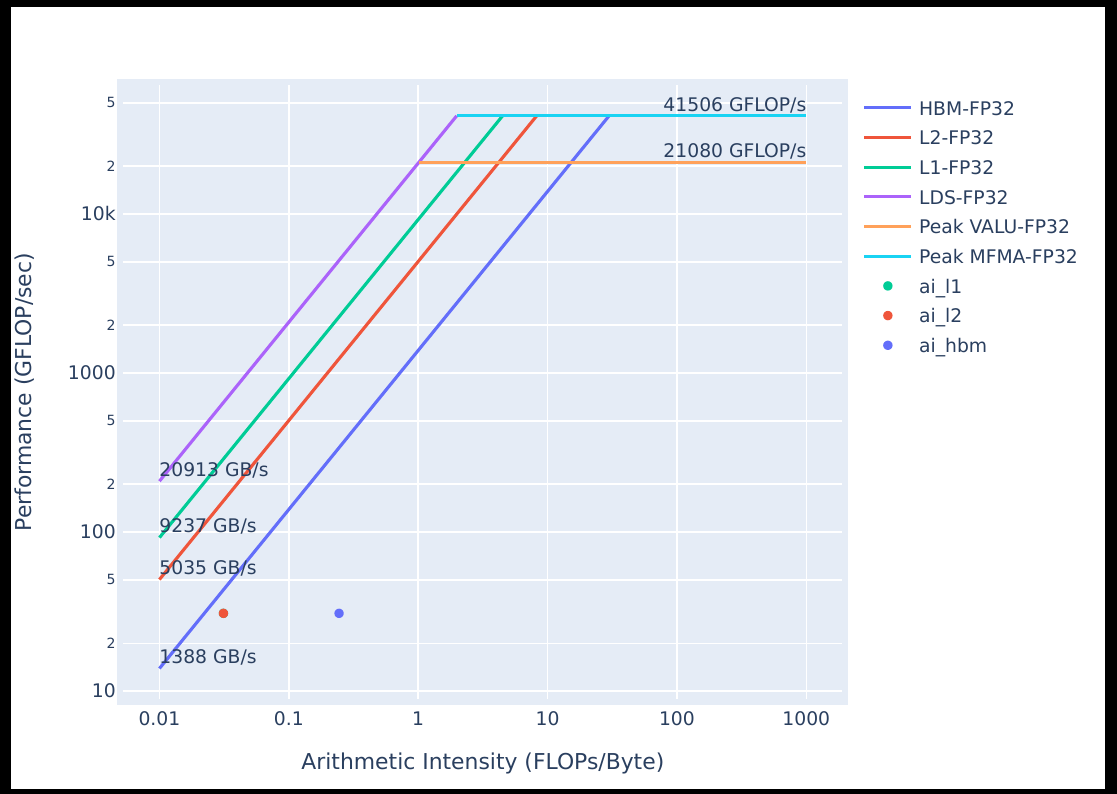
These slides always put `problem` as the baseline, and `solution` as the comparative

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

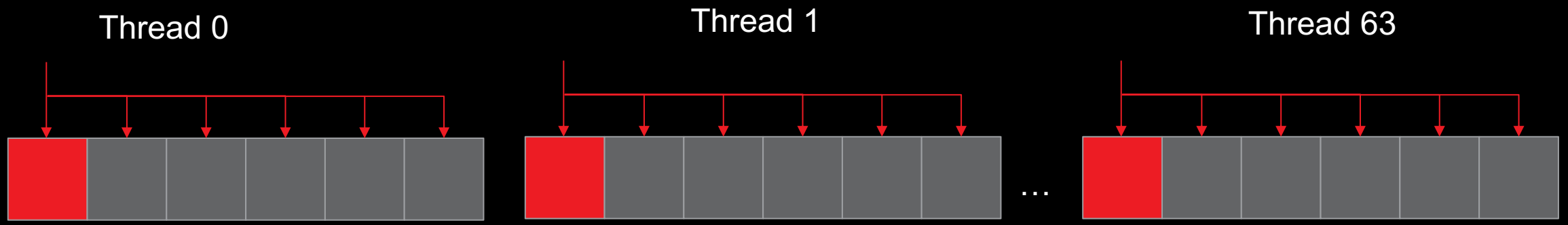
Exercise 3: Register Occupancy Limiter - Takeaways

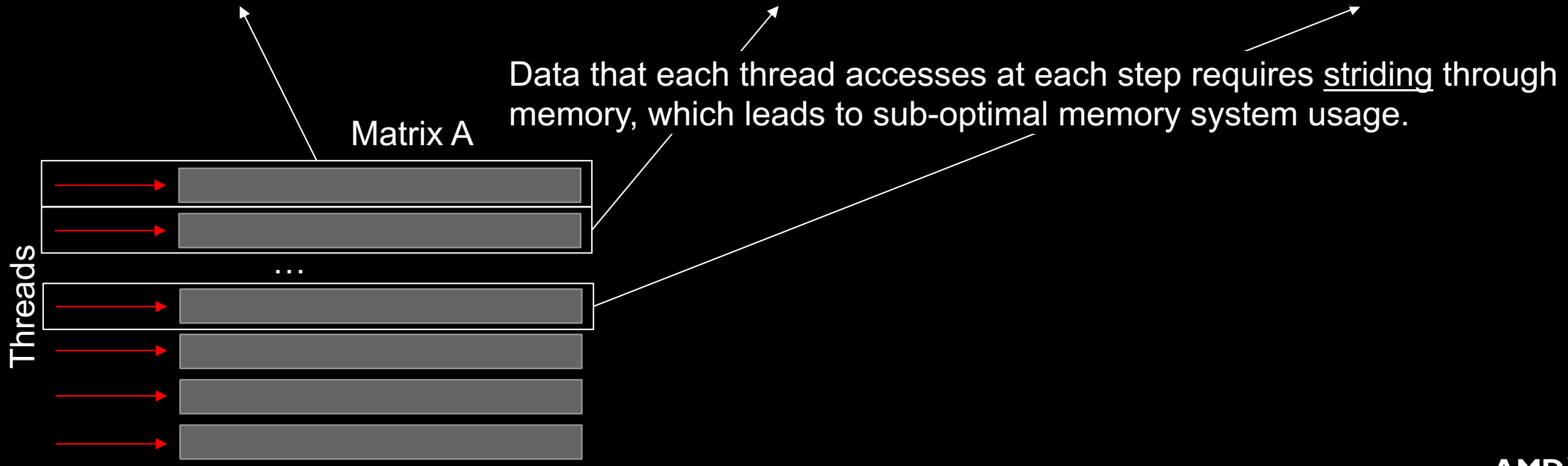
- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - Asserts, and even excessive use of math functions in kernels can degrade performance
- In this case the occupancy limit was very minor, despite a large number in the SPI stat
- AGPR usage in the 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”?

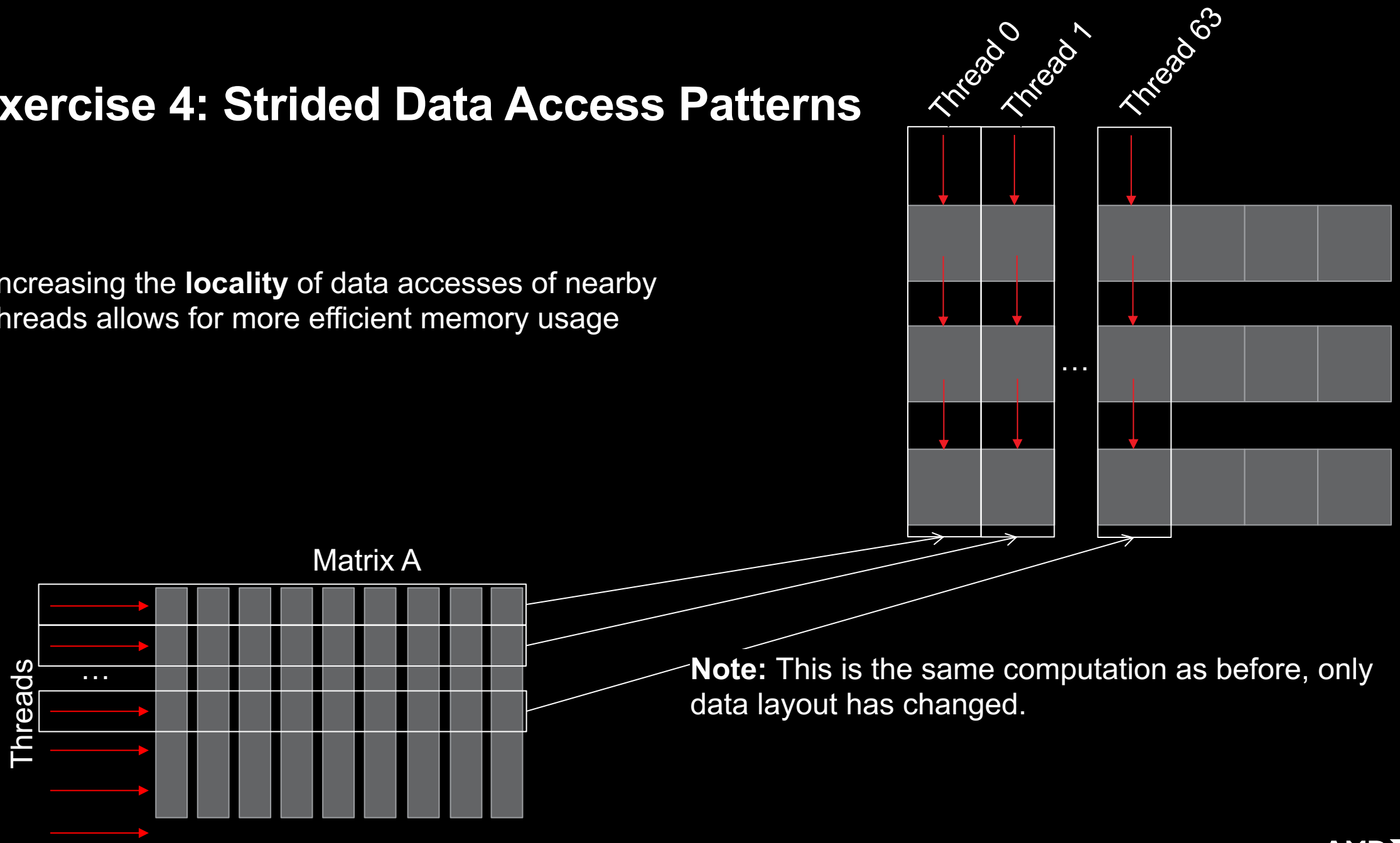




Data that each thread accesses at each step requires striding through memory, which leads to sub-optimal memory system usage.

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

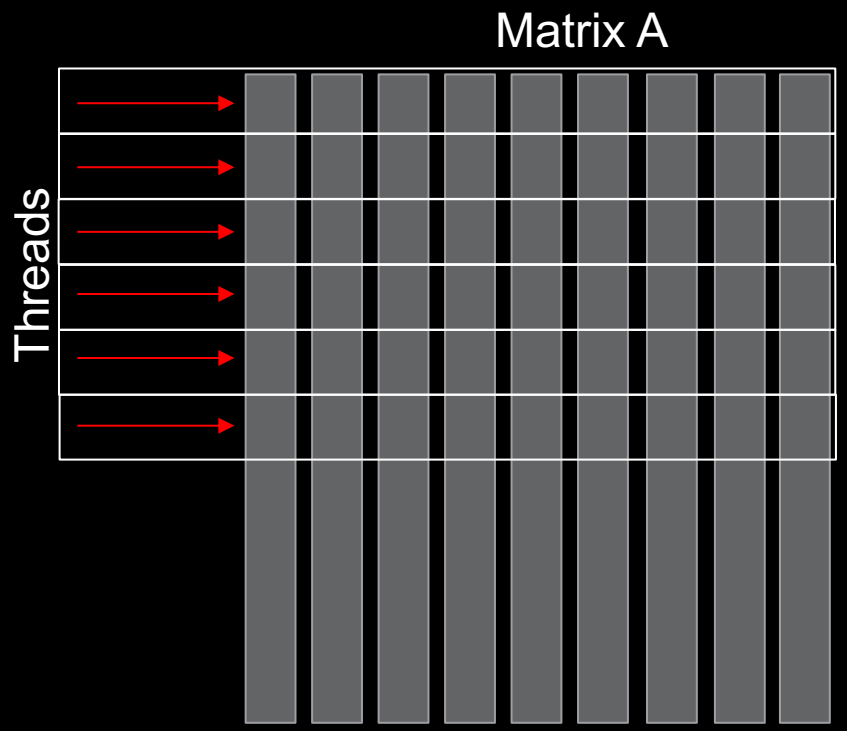
Exercise 4: Omniperf Speed-of-Light Cache Access Statistics

- 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

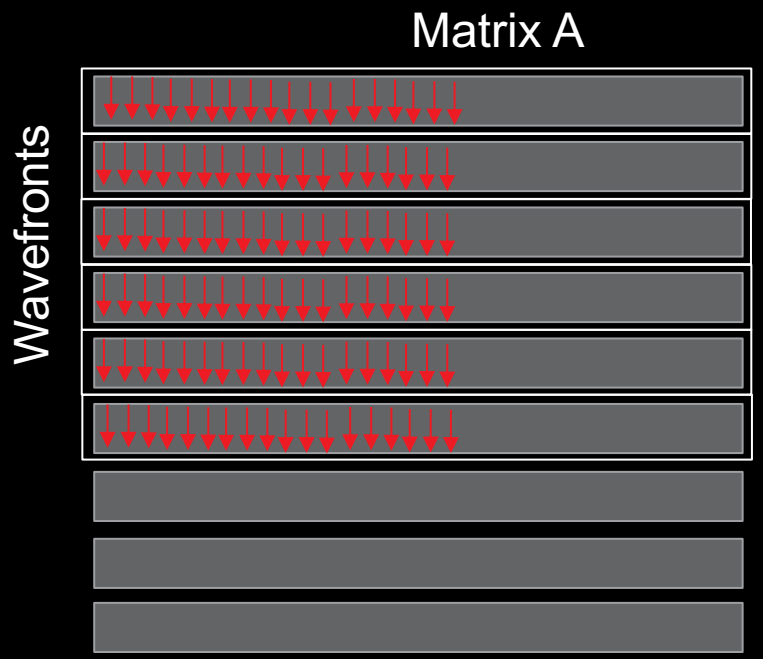
- These types of optimizations are the most difficult to execute
  - Generally, it is difficult to determine if the runtime of one algorithm will be faster than another
- We start with the solution from last exercise as our problem
  - Speed-of-light cache statistics showed that we had ~0% hit rate in 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

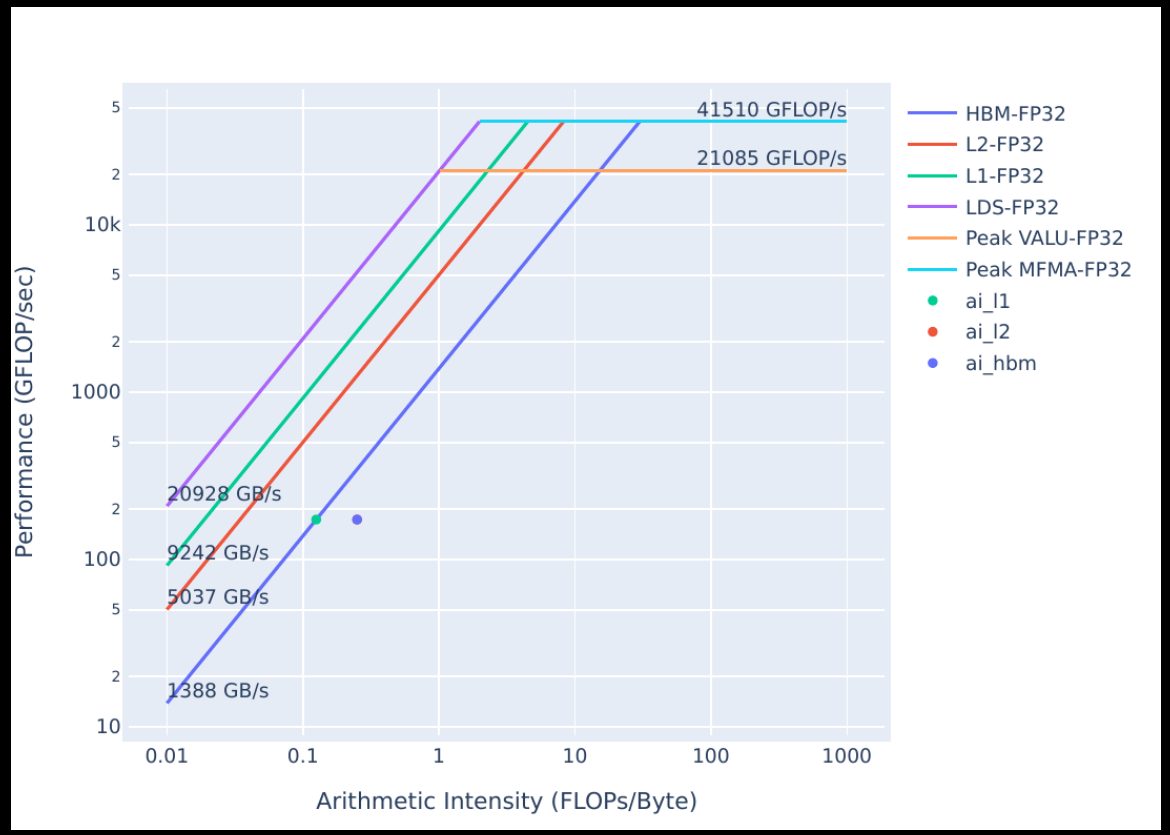
Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
17.3.0	Req	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	Req per wave	- ~53x
...									
17.3.6	Hits	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	Hits per wave	
17.3.7	Misses	32774.06	513.38 (-98.43%)	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

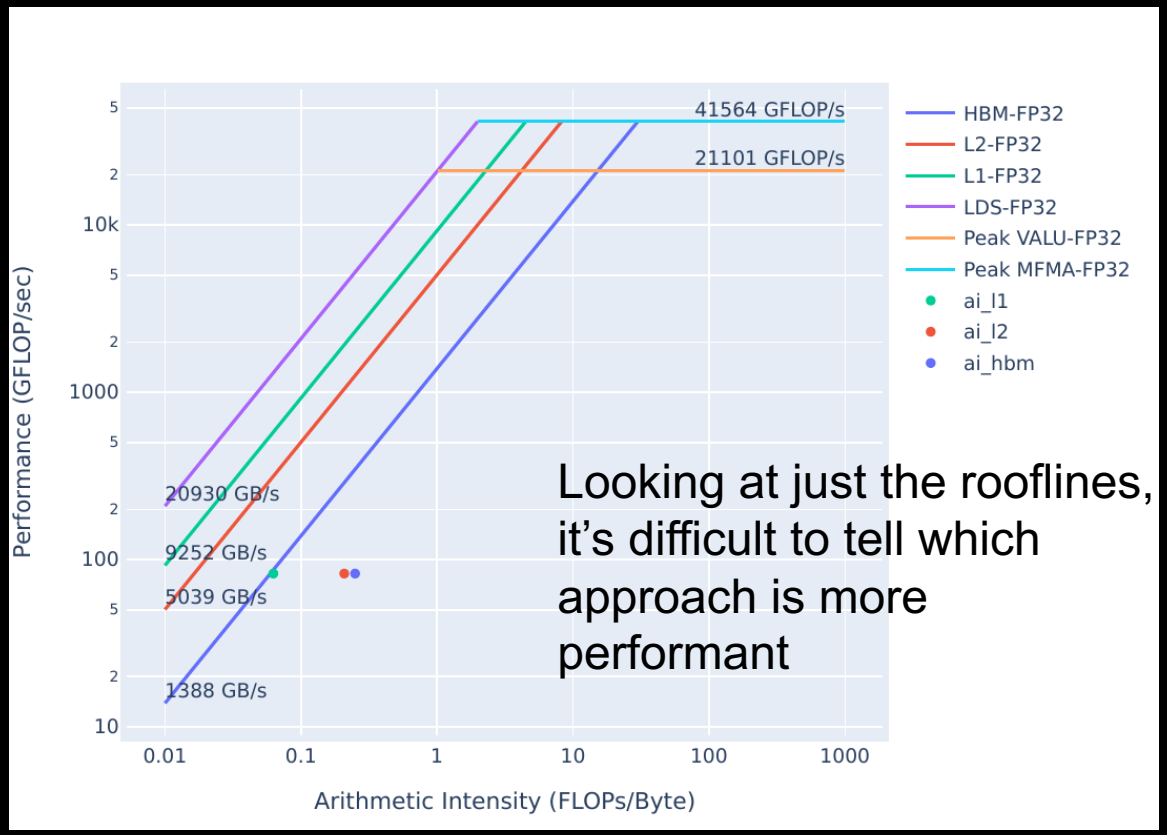
Exercise 5: It Can Be Hard to Compare Rooflines Between Algorithms

- `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

Questions?

DISCLAIMERS AND ATTRIBUTIONS


AMD