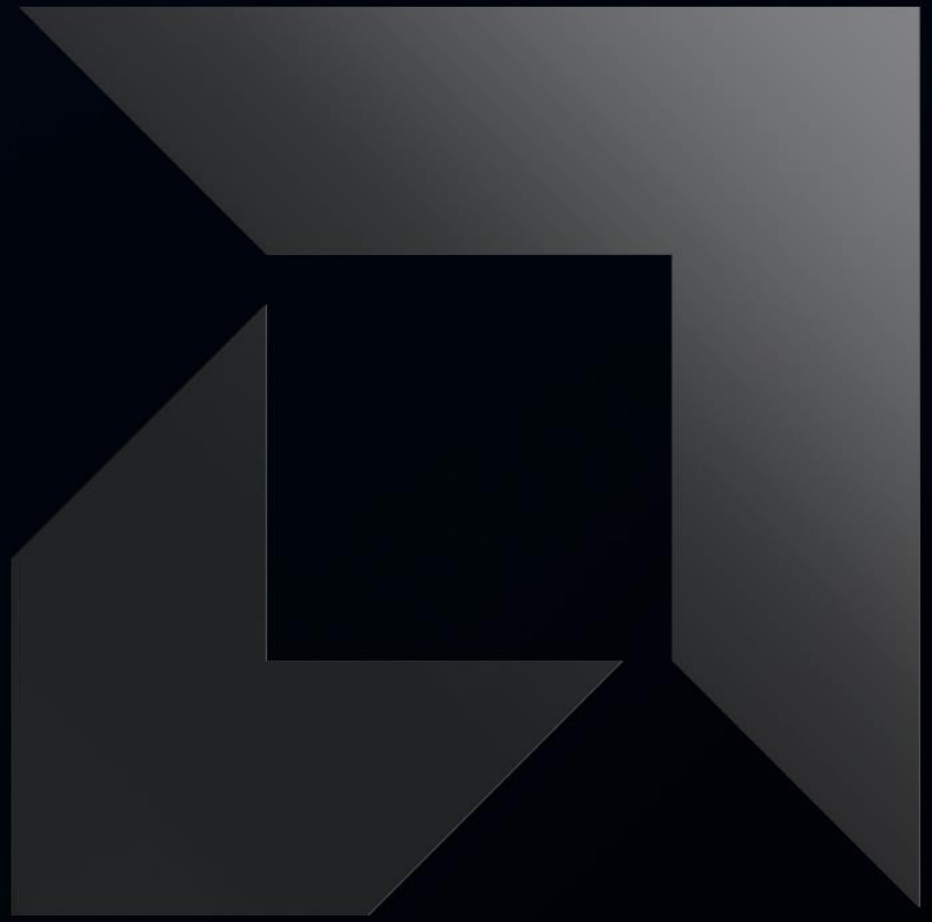


Introduction to Omniperf (rocprofiler-compute)

Presenter: Sam Antao

LUMI pre-hackathon training

May 7th, 2025

AMD

together we advance_

Contributors

- Cole Ramos
- Xiaomin Lu
- Noah Wolfe
- George Markomanolis
- Austin Ellis
- Gina Sitaraman
- Johanna Potyka
- Quentarius Moore

Background – AMD Profilers

ROC-profiler (rocprof)

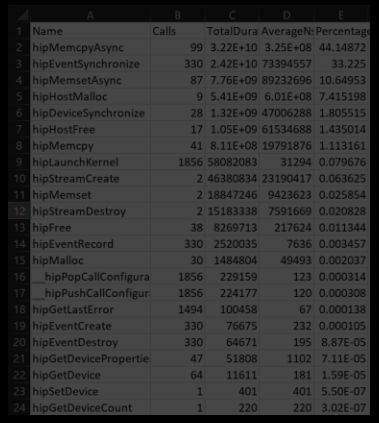
Hardware Counters	Raw collection of GPU counters and traces	
	Counter collection with user input files	Counter results printed to a CSV
Traces and timelines	Trace collection support for	
	CPU copy	HIP API	HSA API	GPU Kernels
Visualisation	Traces visualized with Perfetto	

Omnitrace

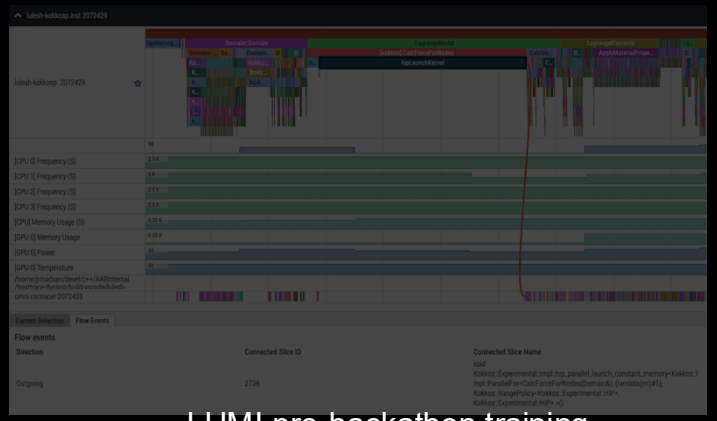
Trace collection	Comprehensive trace collection				
	CPU	GPU			
Supports	CPU copy	HIP API	HSA API	GPU Kernels	
	OpenMP®	MPI	Kokkos	p-threads	multi-GPU
Visualisation	Traces visualized with Perfetto				

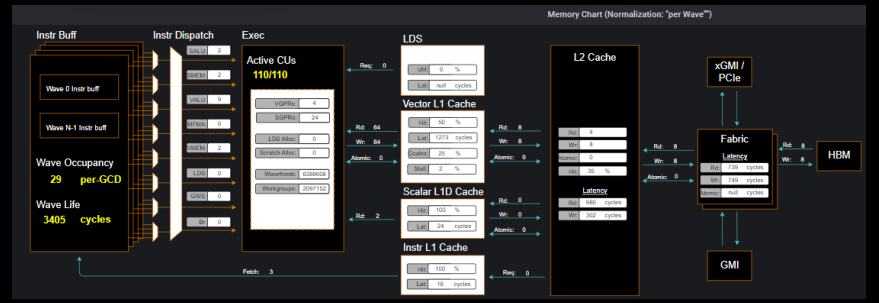
Omniperf

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



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





Open-source Client-side Installation is Easy

Download the latest version from here: https://github.com/ROCM/rocprofiler-compute/releases

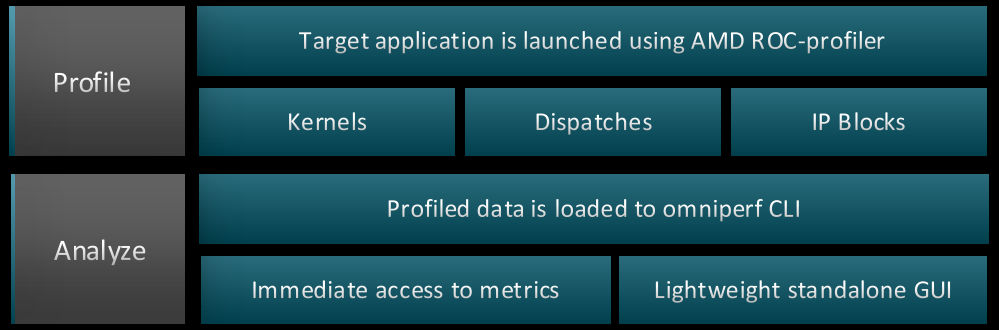
Full documentation: https://rocm.github.io/rocprofiler-compute/

```
wget https://github.com/ROCm/rocprofiler-compute/releases/download/v2.0.1/omniperf-v2.0.1.tar.gz

tar zxvf  omniperf-v2.0.1.tar.gz

cd omniperf-v2.0.1/
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}/2.0.1 \
        -DPYTHON_DEPS=${INSTALL_DIR}/python-libs \
        -DMOD_INSTALL_PATH=${INSTALL_DIR}/modulefiles ..
make install
export PATH=$INSTALL_DIR/2.0.1/bin:$PATH
```

Dependencies

- ROCm (>=6.0.0), Python™ (>=3.8), 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
```

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

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

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

`omniperf profile` Arguments to Reduce Profiling Overhead

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

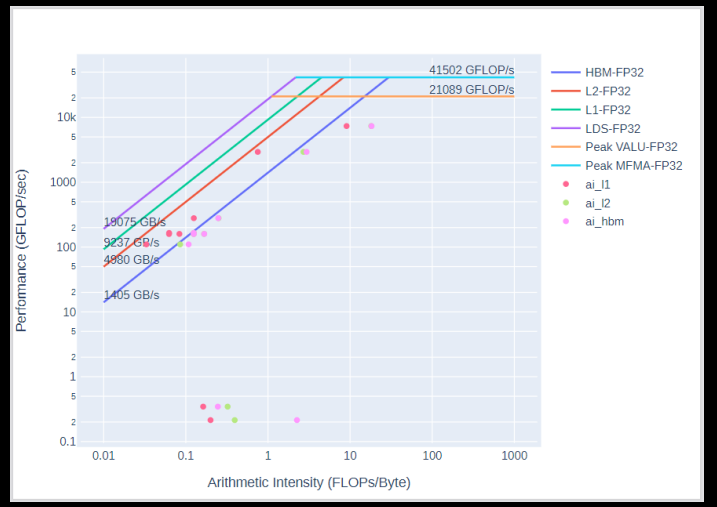
- The `-k` <kernel> flag allows for kernel filtering, which is compatible with the current rocprof utility.
- The `-d` <dispatch> flag allows for dispatch ID filtering, which is compatible with the current rocprof utility.
- The `-b` <ipblocks> allows system profiling on one or more selected IP blocks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

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

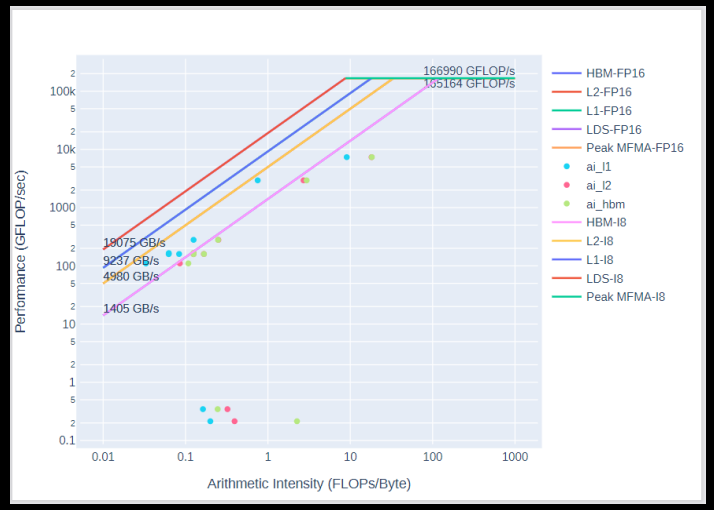
`omniperf profile` Arguments to Generate Roofline PDFs

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

FP32/FP64



FP16/INT8

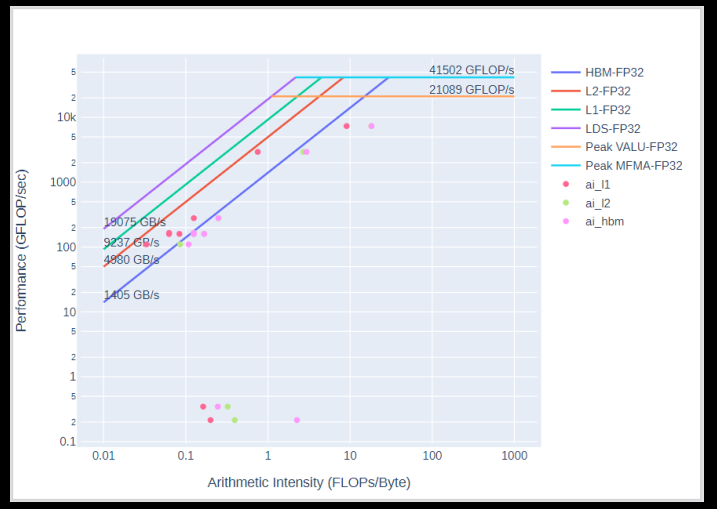


The above plots are saved as PDF output when the --roof-only option is used

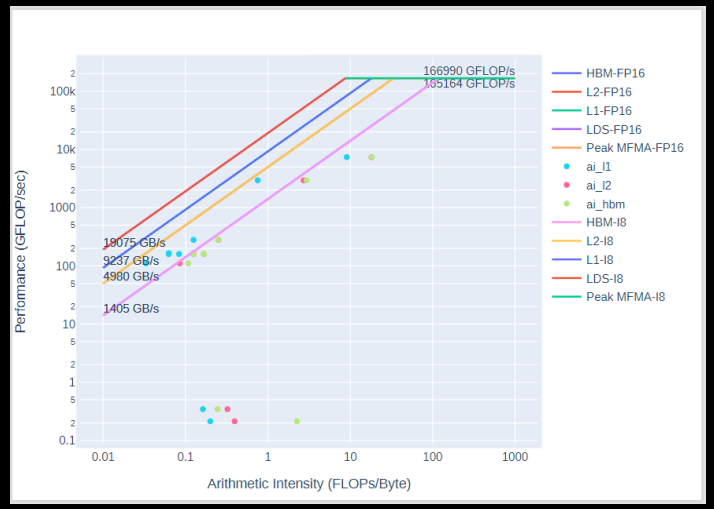
`omniperf profile` Argument to Turn Off Roofline Benchmarking

- Runtime Filtering
  `--kernel`, `--ipblocks`, `--dispatch`
- Standalone Roofline Analysis
  `--roof-only, --kernel-names`
- **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/rocprofiler-compute/omniperf/raw/main/sample/vcopy.cpp
```

Compile with hipcc:

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

Profile with Omniperf:

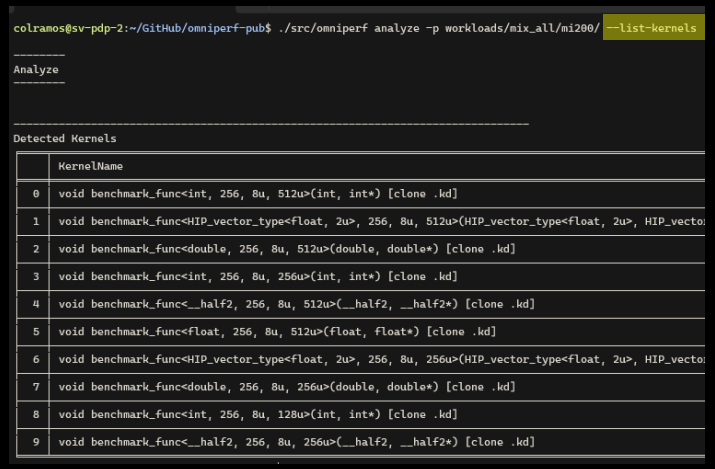
```
$ omniperf profile -n vcopy_all -- ./vcopy 1048576 256
…
-------------
Profile only
-------------

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.

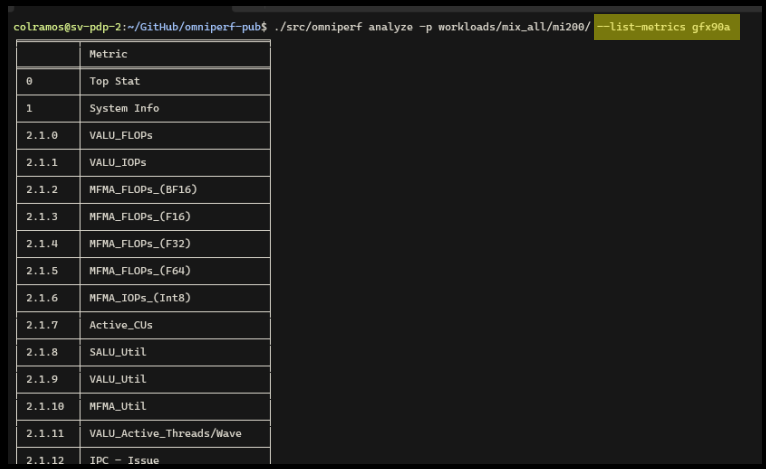
`omniperf analyze` Arguments to Start With

- **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_(8F16)
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

`omniperf analyze` Arguments to Filter Kernels and GPUs

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

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

```
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

`omniperf analyze` Arguments to Compare Workloads

- **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

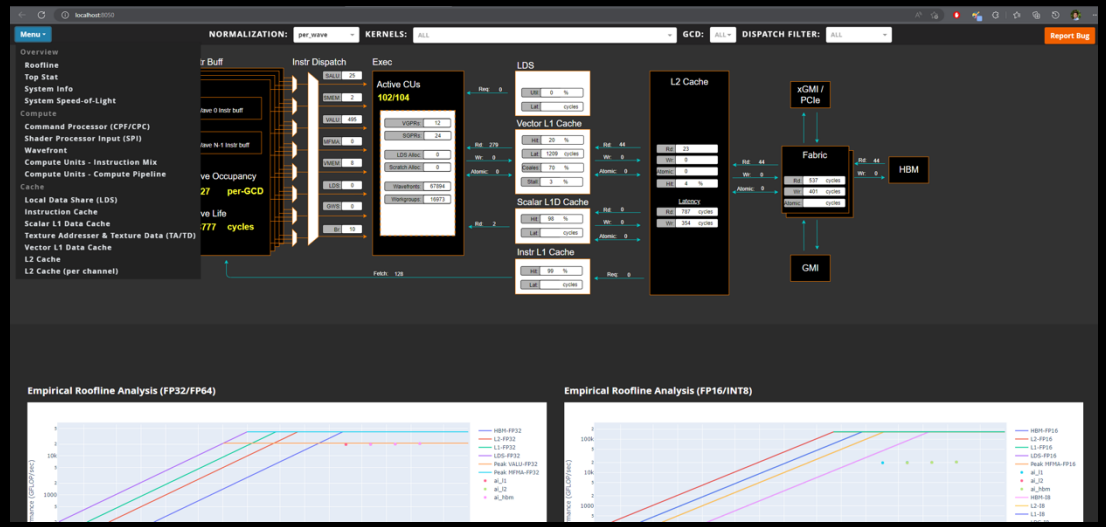
Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit

`omniperf analyze` Argument to Launch Standalone GUI

- Baseline Analysis

  `--path <workload1_path> --path <workload2_path>`
- **Launch a standalone HTML page from terminal**

  `--gui <port>`



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

```
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

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<...

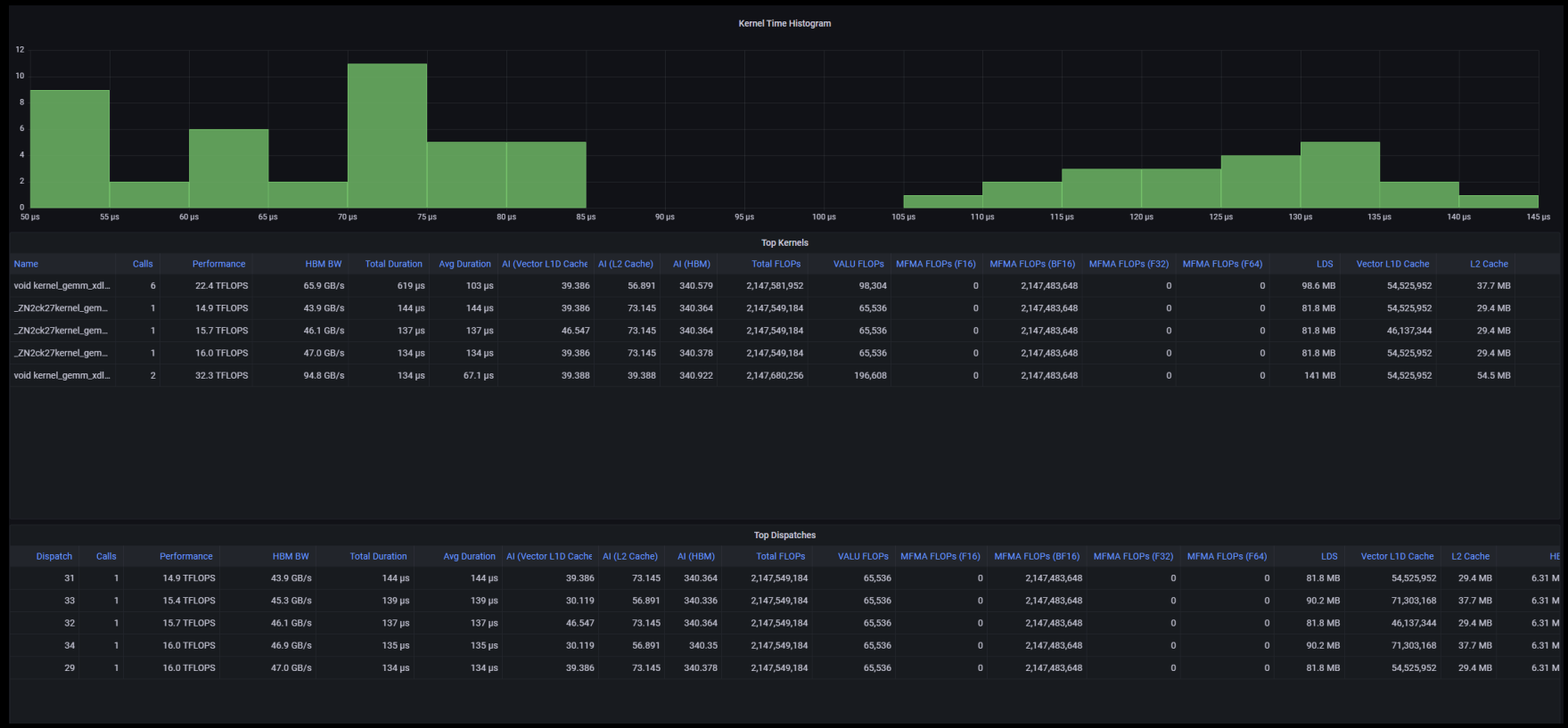
Dispatch IDs - Baseline

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

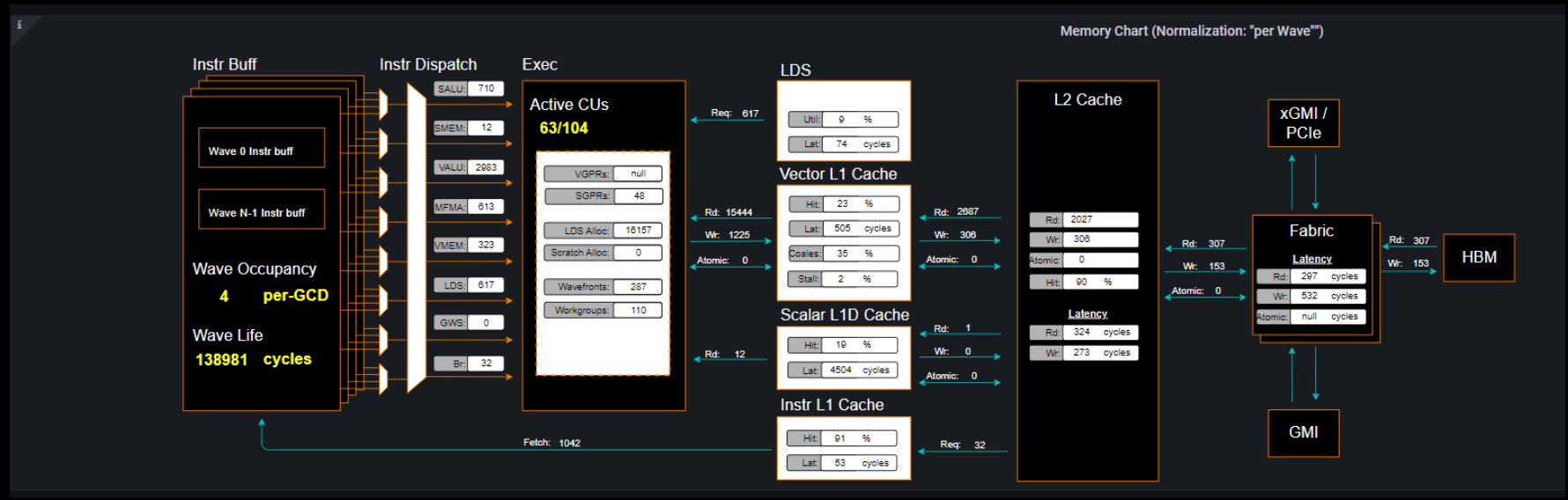
- System Info
- System Speed-of-Light
- Kernel Stats



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

High level Metrics

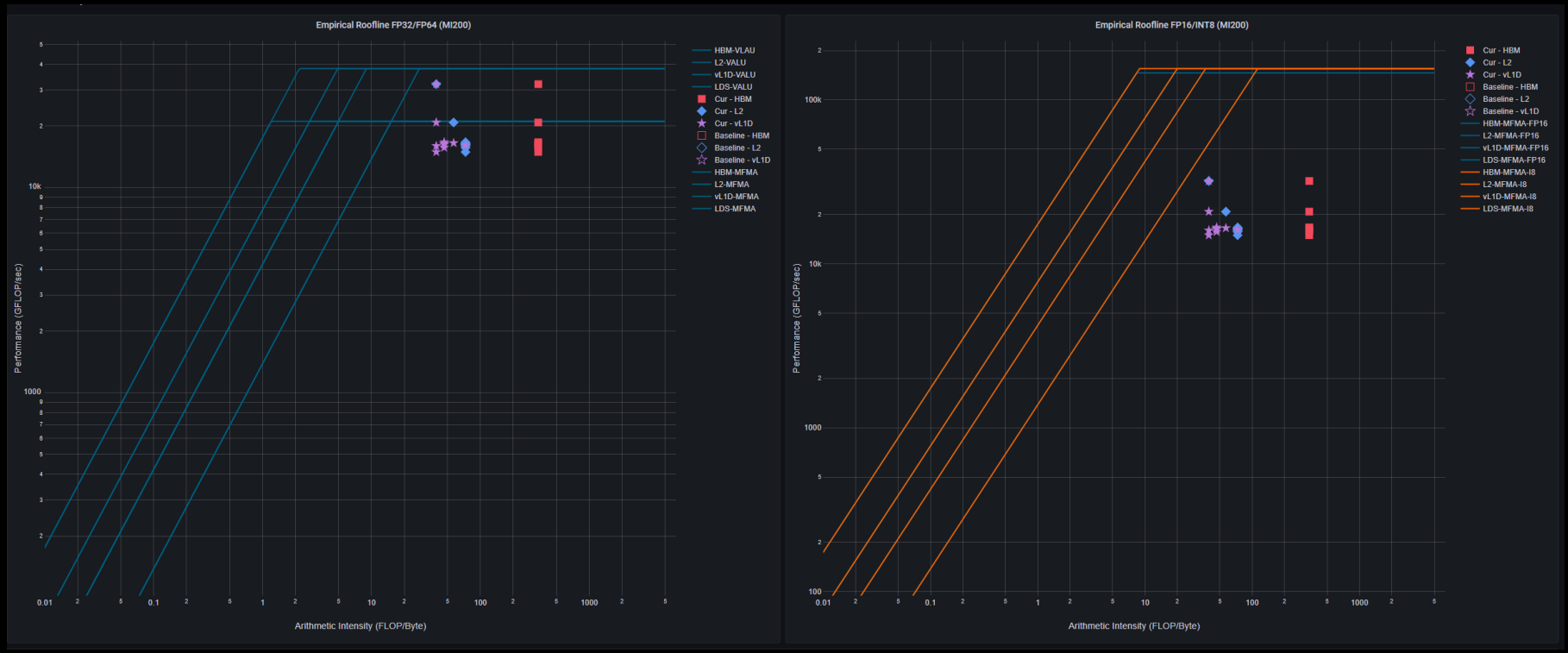
- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis



Illustrate data movement and performance on key components of target architecture

High level Metrics

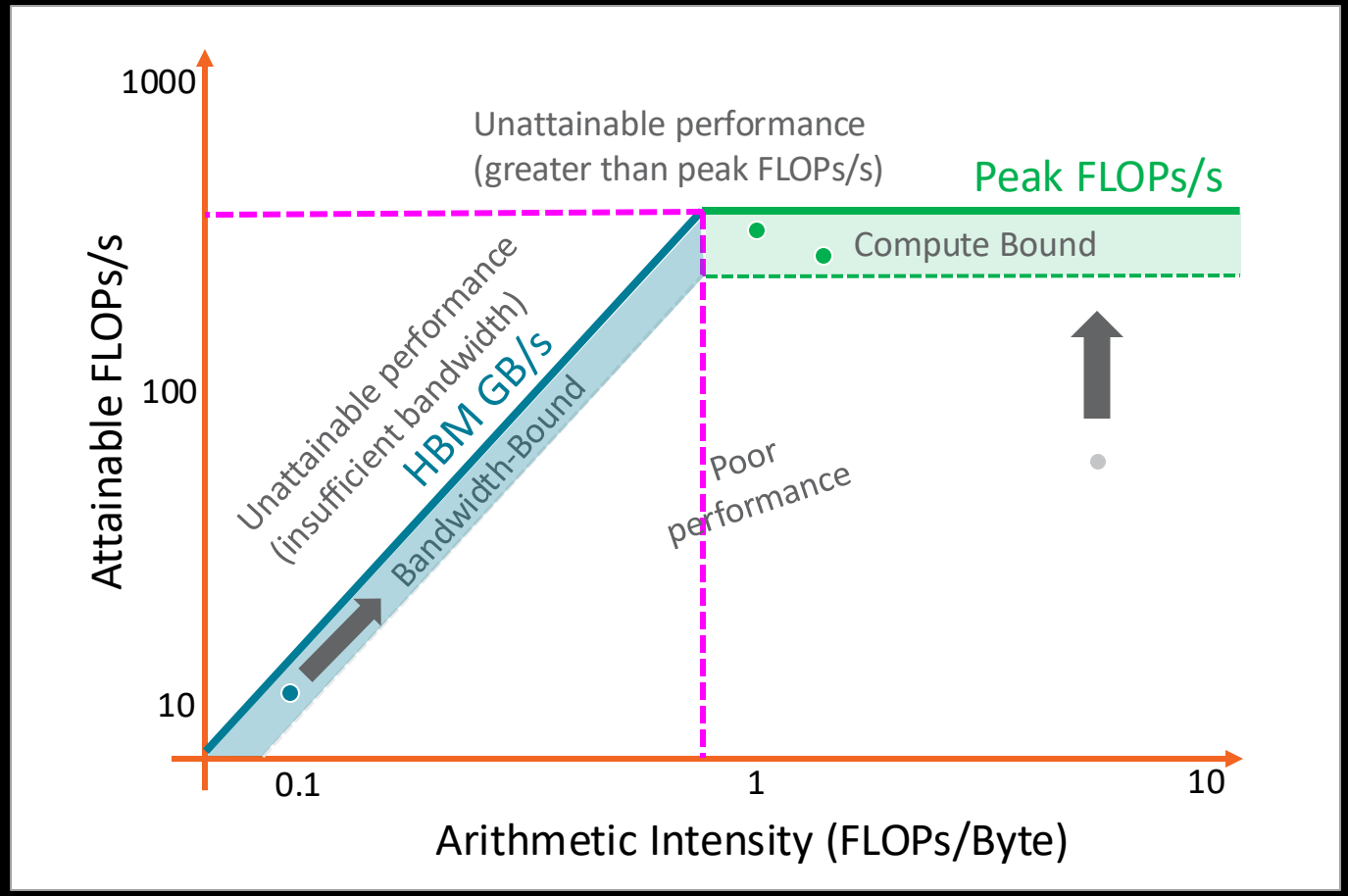
- System Info
- System Speed-of-Light
- Kernel Stats
- Memory Chart Analysis
- Roofline Analysis



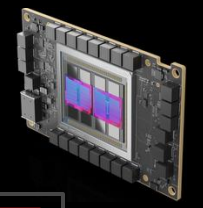
Derived Empirical Roofline analysis broken into two major instruction mixes. Showing application performance relative to measured maximum achievable performance

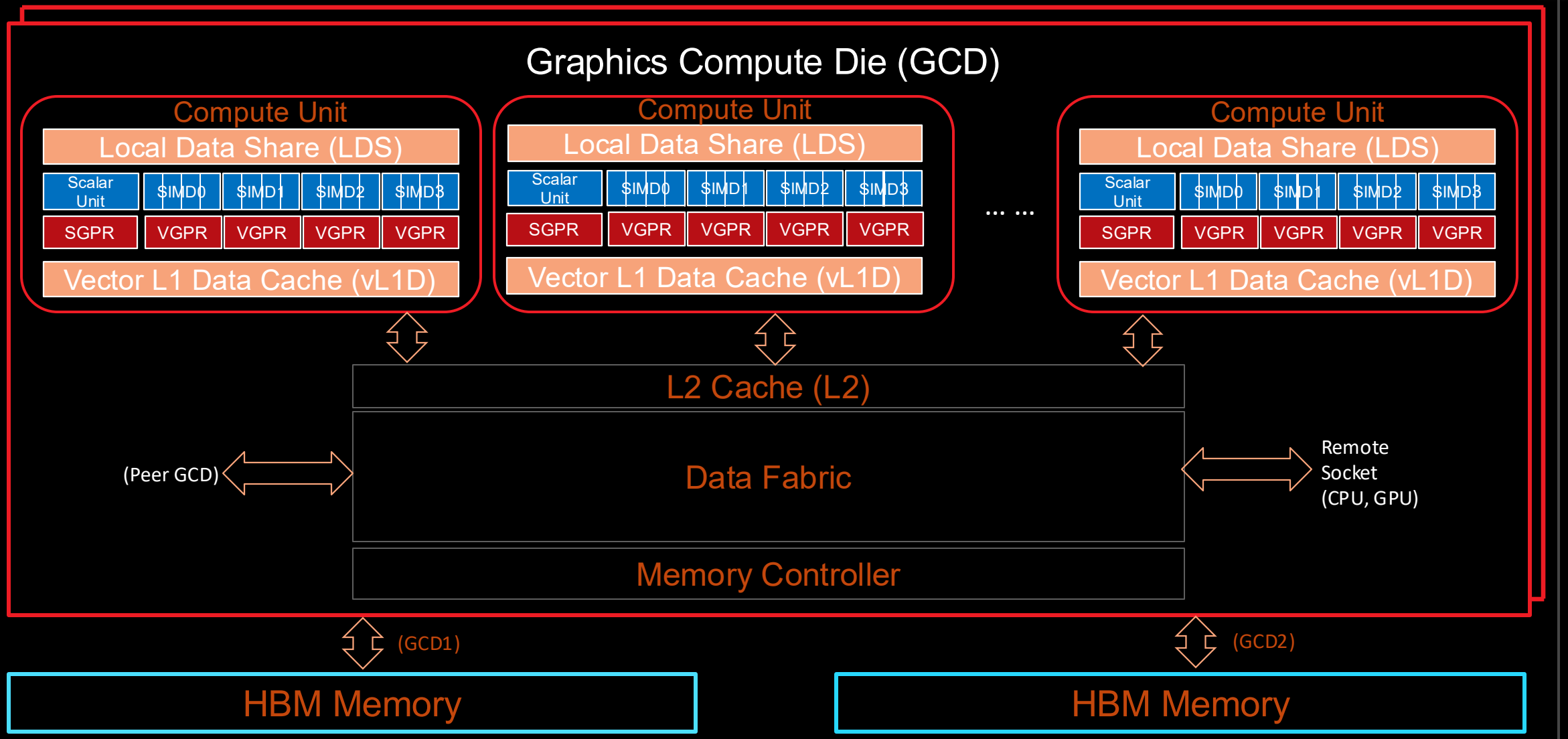
Background – What is roofline?

- 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

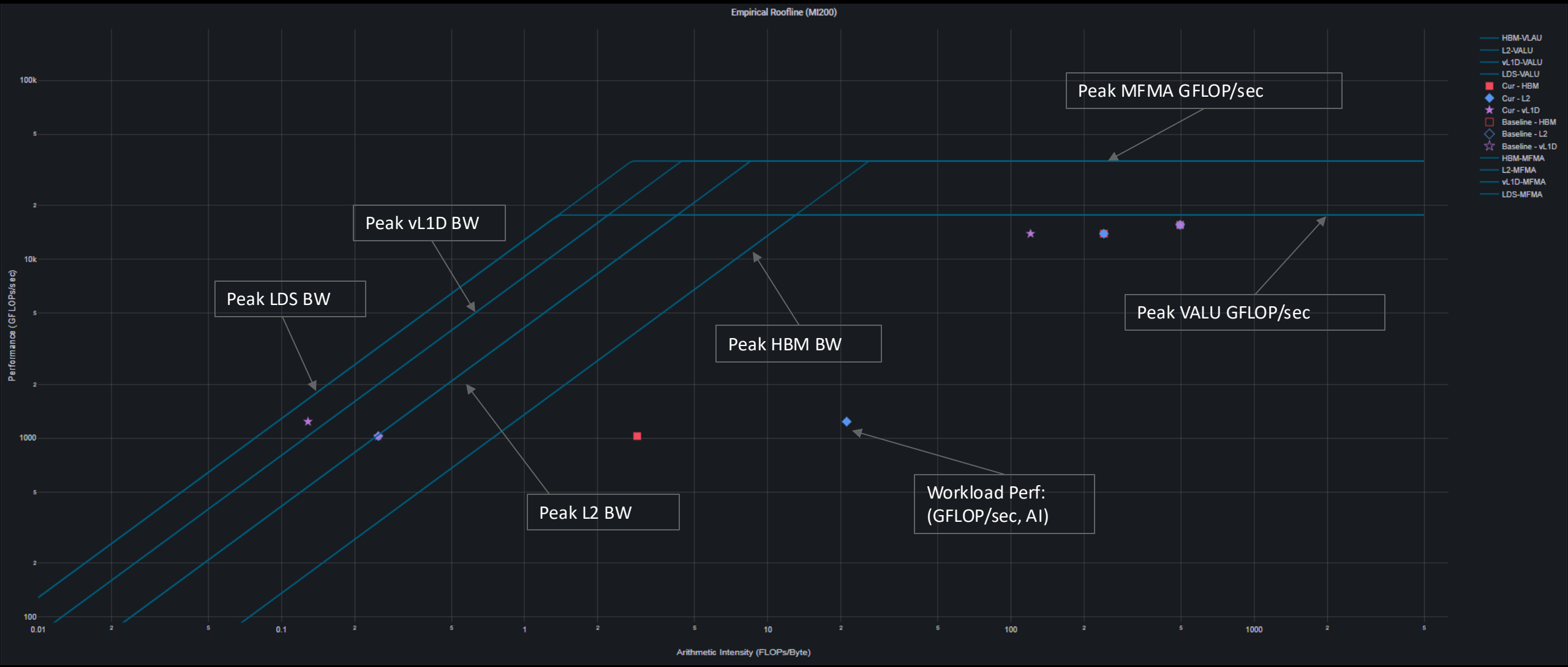


Overview - AMD Instinct™ MI200 Architecture





Empirical Hierarchical Roofline on MI200 - Overview

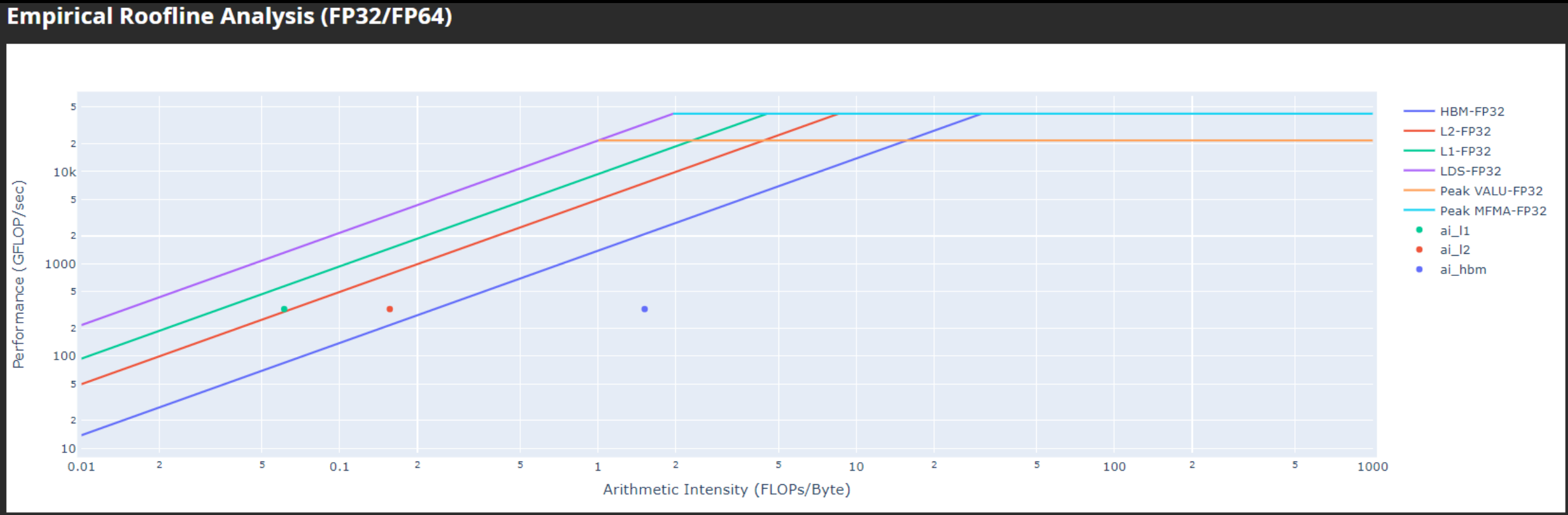


Example – DAXPY with a loop in the kernel

DAXPY – with a loop in the kernel

```cpp
#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



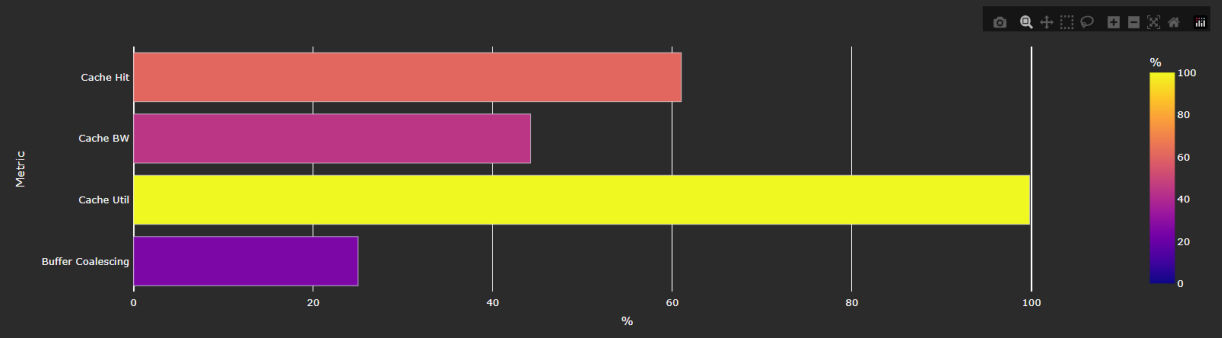
- Performance: almost 330 GFLOPs

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

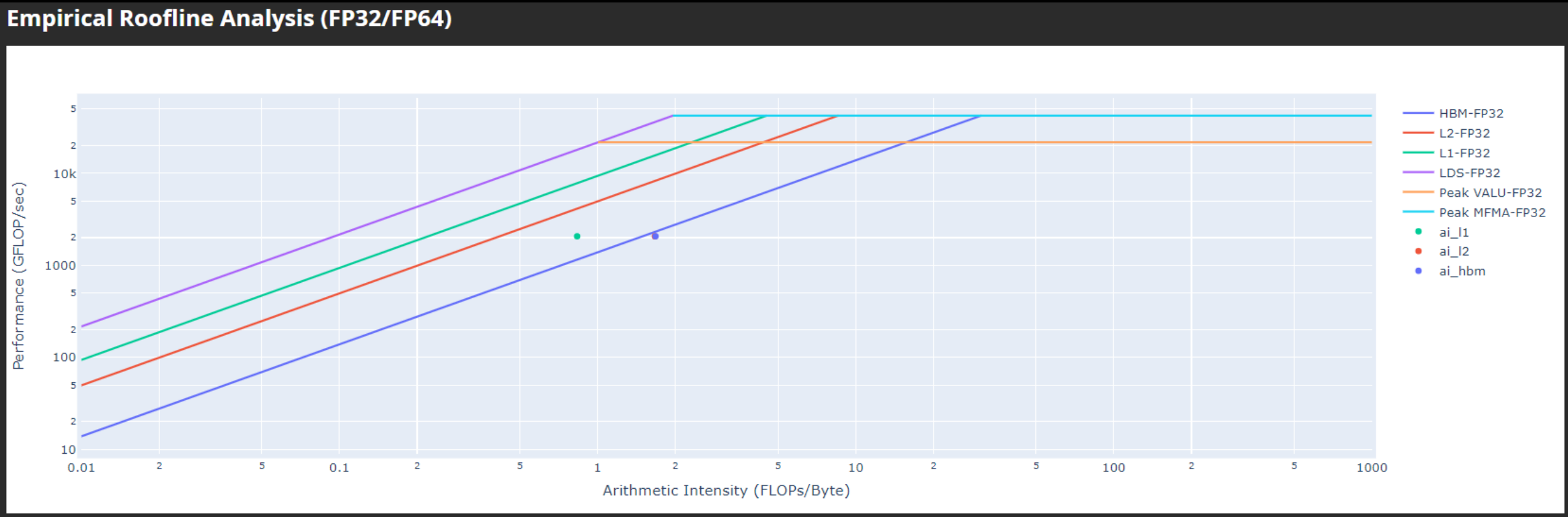
16.3 L1D Cache Accesses

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



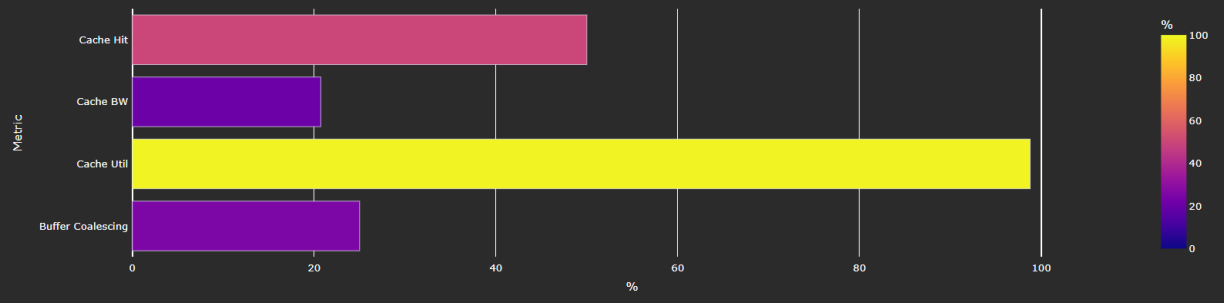
- Performance: almost 2 TFLOPs

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

Summary

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

Questions?

ssh <you user>@lumi.csc.fi

https://hackmd.io/@sfantao/lumi-prehack-may-2025

DISCLAIMERS AND ATTRIBUTIONS


AMD