



# Introduction to Omnipperf

and Hierarchical Roofline on AMD Instinct™ MI200 GPUs

Suyash Tandon, Xiaomin Lu, Noah Wolfe, George Markomanolis, Bob Robey

EuroCC-AMD Workshop  
May 5, 2023

**AMD**   
together we advance\_

# 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

	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

## Omnitrace

**Trace collection**

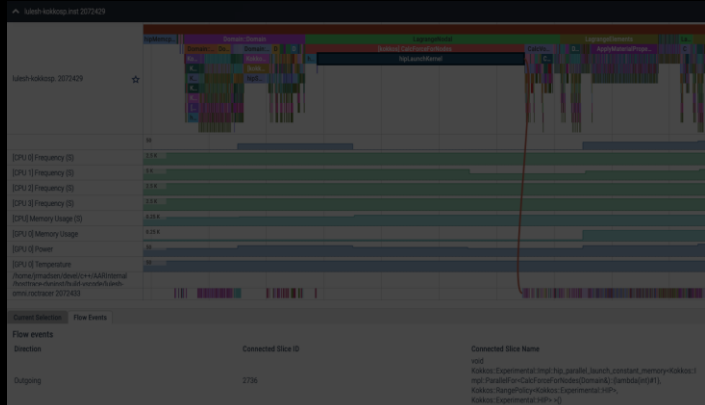
- Comprehensive trace collection
- CPU
- GPU

**Supports**

- CPU copy
- HIP API
- HSA API
- GPU Kernels
- OpenMP®
- MPI
- Kokkos
- p-threads
- multi-GPU

**Visualisation**

- Traces visualized with Perfetto



## Omniperf

**Performance Analysis**

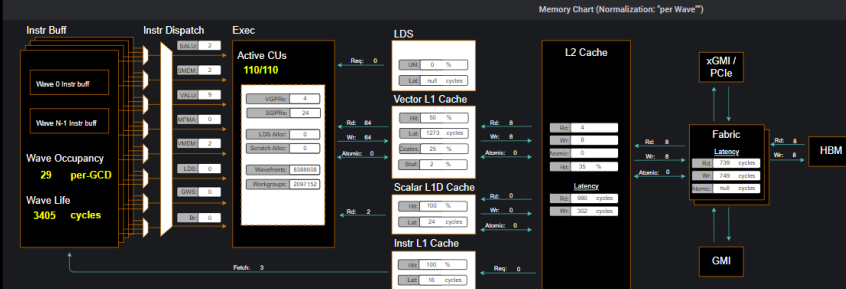
- Automated collection of hardware counters
- Analysis
- Visualisation

**Supports**

- Speed of Light
- Memory chart
- Rooflines
- Kernel comparison

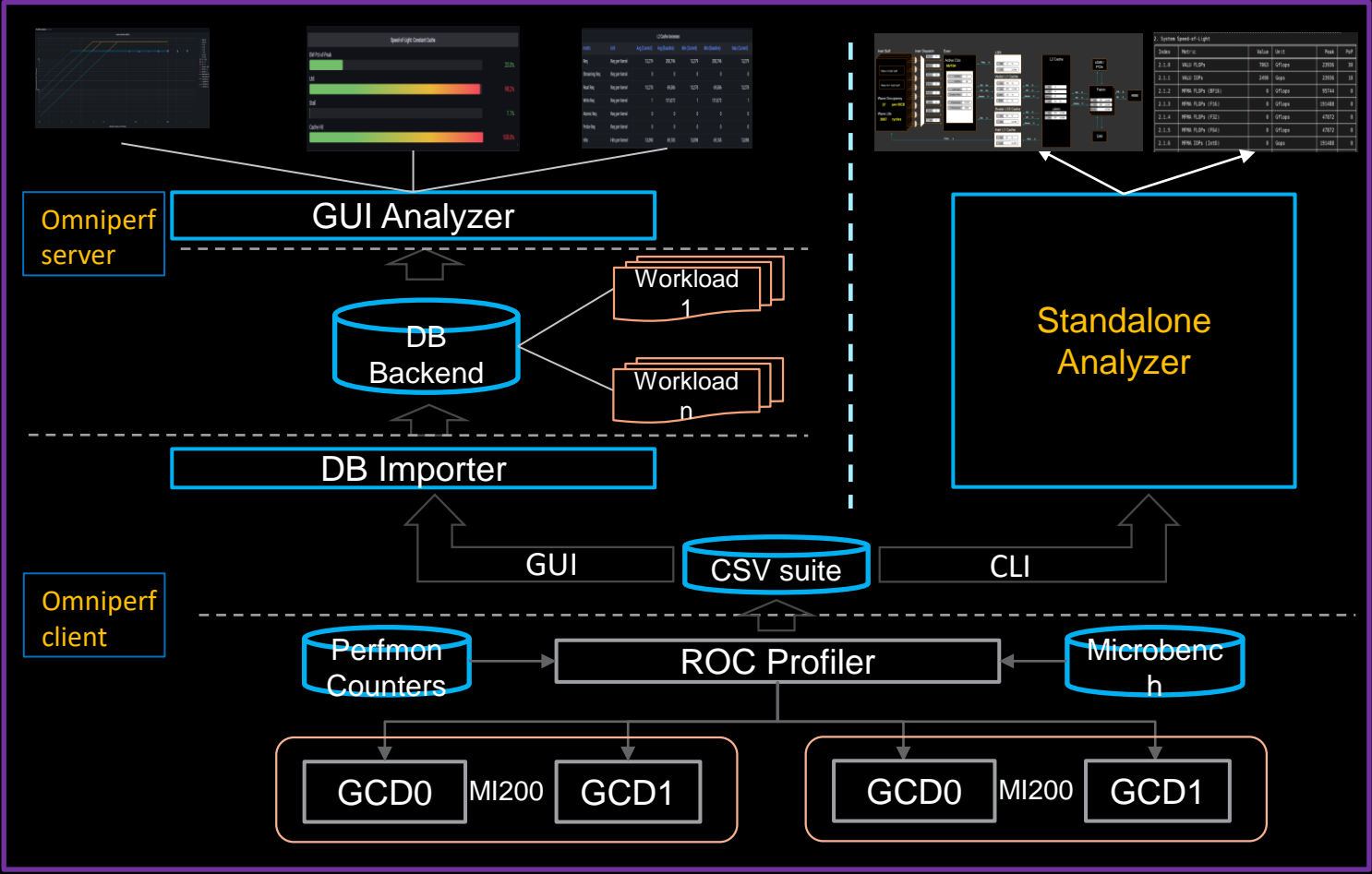
**Visualisation**

- With Grafana or standalone GUI



# Omniperf: Automated Collection of Hardware Counters and Analysis

AMD Research Tool	Repository: <a href="https://github.com/AMDResearch/omniperf">https://github.com/AMDResearch/omniperf</a>			
	<b>X</b> Not part of ROCm stack	Built on top of ROC-profiler		
Integrated Performance Analyzer for AMD GPUs	Speed-of-Light	Roofline	Memory chart	Baseline comparison
	Sub-system performance analysis			
	LDS	vL1D	L2 Cache	HBM
	Shader Compute	Wavefront	Instruction mix	Latencies
INSTINCT™ Support	MI200		MI100	
User Interfaces	Grafana™ GUI	Standalone GUI	Command Line (CLI)	



Refer to [current documentation](#) for recent updates

# Omniperf features

Omniperf Features	
MI200 support	Roofline Analysis Panel ( <i>Supported on MI200 only, SLES 15 SP3 or RHEL8</i> )
MI100 support	Command Processor (CP) Panel
Standalone GUI Analyzer	Shader Processing Input (SPI) Panel
Grafana/MongoDB GUI Analyzer	Wavefront Launch Panel
Dispatch Filtering	Compute Unit - Instruction Mix Panel
Kernel Filtering	Compute Unit - Pipeline Panel
GPU ID Filtering	Local Data Share (LDS) Panel
Baseline Comparison	Instruction Cache Panel
Multi-Normalizations	Scalar L1D Cache Panel
System Info Panel	Texture Addresser and Data Panel
System Speed-of-Light Panel	Vector L1D Cache Panel
Kernel Statistic Panel	L2 Cache Panel
Memory Chart Analysis Panel	L2 Cache (per-Channel) Panel

# Omniperf

- Omniperf is an integrated performance analyzer for AMD GPUs built on ROCprofiler
- Omniperf executes the code many times to collect various hardware counters (over 100 counters default behavior)
- Using specific filtering options (kernel, dispatch ID, metric group), the overhead of profiling can be reduced
- Roofline analysis is supported on MI200 GPUs
- Omniperf shows many panels of metrics based on hardware counters, we will show a few here
- Typical Omniperf workflows:
  - Profile + Analyze with CLI or visualize with standalone GUI
  - Profile + Import to database and visualize with Grafana
- Omniperf targets MI100 and MI200 and future generation AMD GPUs
- Omniperf requires to use just 1 MPI process
- For problems, create an issue here: <https://github.com/AMDResearch/omniperf/issues>

# Client-side installation (if required)

 Download the latest version from here: <https://github.com/AMDRResearch/omniperf/releases>

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

```
wget https://github.com/AMDRResearch/omniperf/releases/download/v1.0.8-PR2/omniperf-v1.0.8-PR2.tar.gz

tar zxvf omniperf-v1.0.8-PR2.tar.gz

cd omniperf-v1.0.8-PR2/
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}/1.0.8 \
      -DPYTHON_DEPS=${INSTALL_DIR}/python-libs \
      -DMOD_INSTALL_PATH=${INSTALL_DIR}/modulefiles ..
make install
export PATH=${INSTALL_DIR}/1.0.8/bin:$PATH
```

# Omniperf modes

Profile	Target application is launched using AMD ROC-profiler		
	Kernels	Dispatches	IP Blocks
Analyze	Profiled data is loaded to omniperf CLI		
	Immediate access to metrics	Lightweight standalone GUI	
Database	Profiled data is imported to Grafana™ database		
	Grafana™ GUI is based on MongoDB	Interact with saved workload database	

## Basic command-line syntax:

### Profile:

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

### Analyze:

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

To use a lightweight standalone GUI with CLI analyzer:

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

### Database:

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

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

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

For problems, create an issue here: <https://github.com/AMDResearch/omniperf/issues>

Documentation: <https://amdresearch.github.io/omniperf>

# Omniperf profiling

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

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

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

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

```
...
```

```
-----  
Profile only  
-----
```

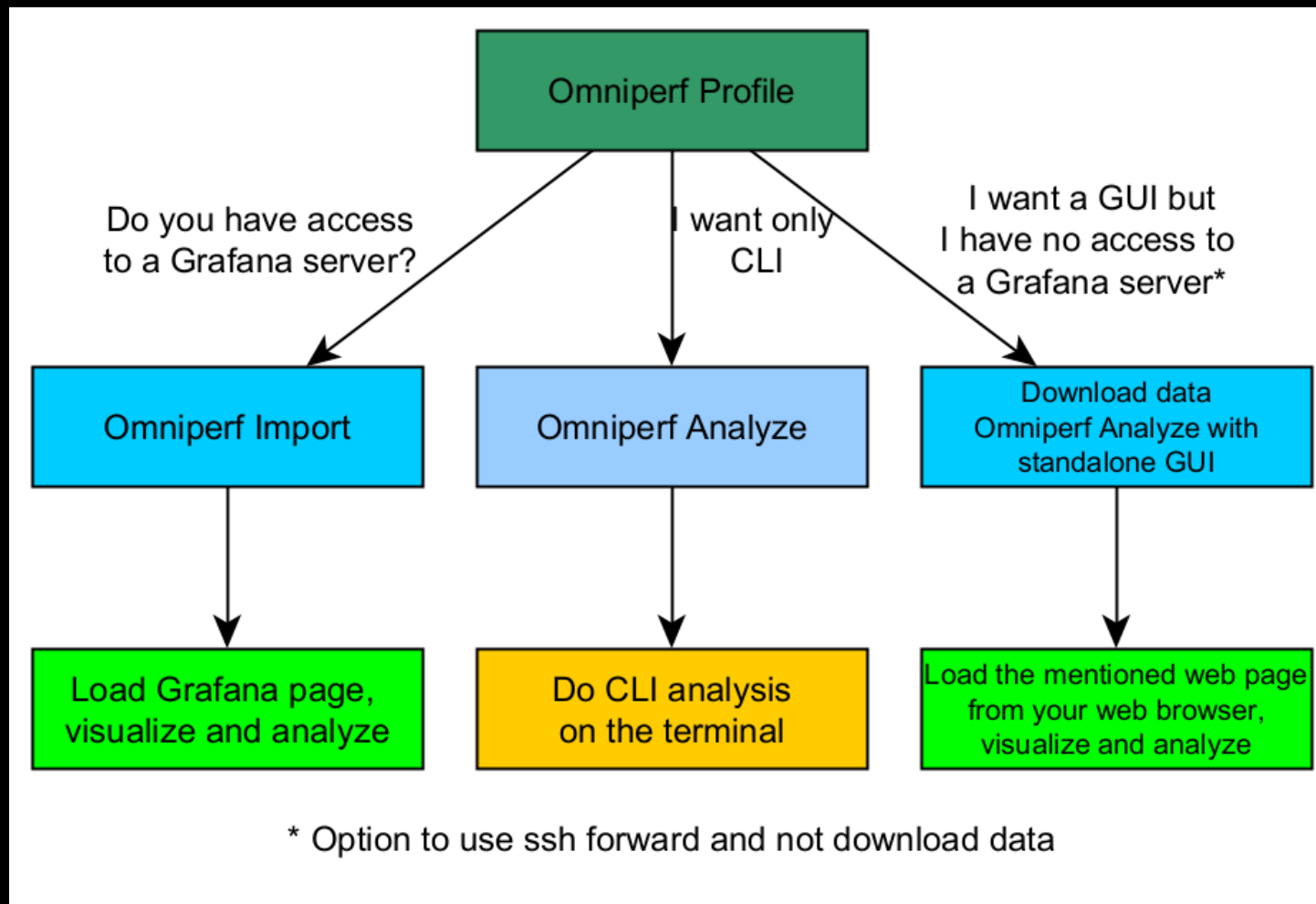
```
omniperf ver: 1.0.4  
Path: /pfs/lustrep4/scratch/project_462000075/markoman/omniperf-  
1.0.4/build/workloads  
Target: mi200  
Command: ./vcopy 1048576 256  
Kernel Selection: None  
Dispatch Selection: None  
IP Blocks: All
```

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 workflows



# Omniperf analyze

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

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

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

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

A new directory will be created called workloads/vcopy\_all

Analyze the profiled workload:

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

0. Top Stat

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pc
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	341123.00	341123.00	341123.00	100.00

## 2. System Speed-of-Light

Index	Metric	Value	Unit	Peak	PoP
2.1.0	VALU FLOPs	0.00	Gflop	23936.0	0.0
2.1.1	VALU IOPs	89.14	Giop	23936.0	0.37242200388114116
2.1.2	MFMA FLOPs (BF16)	0.00	Gflop	95744.0	0.0
2.1.3	MFMA FLOPs (F16)	0.00	Gflop	191488.0	0.0
2.1.4	MFMA FLOPs (F32)	0.00	Gflop	47872.0	0.0
2.1.5	MFMA FLOPs (F64)	0.00	Gflop	47872.0	0.0
2.1.6	MFMA IOPs (Int8)	0.00	Giop	191488.0	0.0
2.1.7	Active CUs	58.00	Cus	110	52.72727272727273
2.1.8	SALU Util	3.69	Pct	100	3.6862586934167525
2.1.9	VALU Util	5.90	Pct	100	5.895531580380328
2.1.10	MFMA Util	0.00	Pct	100	0.0
2.1.11	VALU Active Threads/Wave	32.71	Threads	64	51.10526315789473
2.1.12	IPC = Issue	0.08	Insts/cycle	5	10.576640821020212

## 7.1 Wavefront Launch Stats

Index	Metric	Avg	Min	Max	Unit
7.1.0	Grid Size	1048576.00	1048576.00	1048576.00	Work items
7.1.1	Workgroup Size	256.00	256.00	256.00	Work items
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts
7.1.3	Saved Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.4	Restored Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.5	VGPRs	44.00	44.00	44.00	Registers
7.1.6	SGPRs	48.00	48.00	48.00	Registers
7.1.7	LDS Allocation	0.00	0.00	0.00	Bytes
7.1.8	Scratch Allocation	16496.00	16496.00	16496.00	Bytes

# Omniperf Analyze

- Execute omniperf analyze -h to see various options
- Use specific IP block (-b)

## Top kernels:

```
$ srun -n 1 --gpus 1 omniperf analyze -p workloads/vcopy_all/mi200/ -b 0
```

## IP Block of wavefronts

```
$ srun -n 1 --gpus 1 omniperf analyze -p workloads/vcopy_all/mi200/ -b 7.1.2
```

### 0. Top Stat

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	20960.00	20960.00	20960.00	100.00

### 7. Wavefront

#### 7.1 Wavefront Launch Stats

Index	Metric	Avg	Min	Max	Unit
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts

# Omniperf analyze

To see available options and usage instructions:

```
$ omniperf analyze -h
...

Help:
  -h, --help            show this help message and exit

General Options:
  -v, --version          show program's version number and exit
  -V, --verbose          Increase output verbosity

Analyze Options:
  -p [ ...], --path [ ...]  Specify the raw data root dirs or desired results directory.
  -o, --output            Specify the output file.
  --list-kernels          List kernels. Top 10 kernels sorted by duration (descending order).
  --list-metrics          List metrics can be customized to analyze on specific arch:
                          gfx906
                          gfx908
                          gfx90a
  -b [ ...], --metric [ ...] Specify IP block/metric id(s) from --list-metrics for filtering.
  -k [ ...], --kernel [ ...] Specify kernel id(s) from --list-kernels for filtering.
  --dispatch [ ...]       Specify dispatch id(s) for filtering.
  --gpu-id [ ...]         Specify GPU id(s) for filtering.
  -n, --normal-unit       Specify the normalization unit: (DEFAULT: per_wave)
                          per_wave
                          per_cycle
                          per_second
                          per_kernel
  --config-dir            Specify the directory of customized configs.
  -t, --time-unit         Specify display time unit in kernel top stats: (DEFAULT: ns)
                          s
                          ms
                          us
                          ns
  --decimal               Specify the decimal to display. (DEFAULT: 2)
  --cols [ ...]           Specify column indices to display.
  -g                       Debug single metric.
  --dependency            List the installation dependency.
  --gui [GUI]             Activate a GUI to interate with Omniperf metrics.
                          Optionally, specify port to launch application (DEFAULT: 8050)
```

# Easy things you can check

- Are all the CUs being used?
  - If not, more parallelism is required (for most of the cases)
- Are all the VGPRs being spilled?
  - Try smaller workgroup sizes
- Is the code Integer limited?
  - Try reducing the integer ops, usually in the index calculation

# Omniperf analyze with standalone GUI

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

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

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

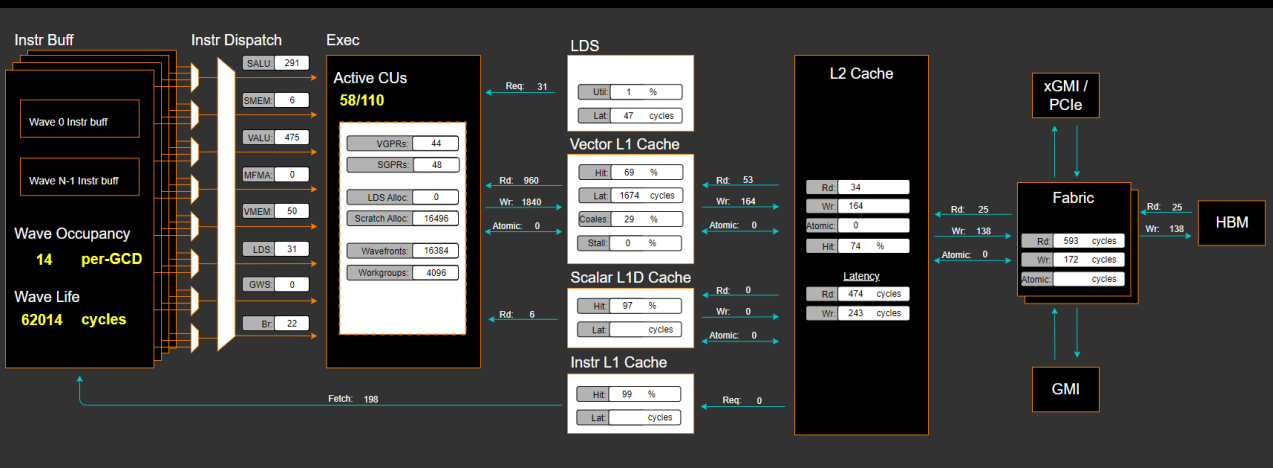
```
$ omniperf profile -n vcopy_all -- ./vcopy 1048576 256
```

A new directory will be created called workloads/vcopy\_all

Analyze the profiled workload:

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

Open web page <http://IP:8050/>

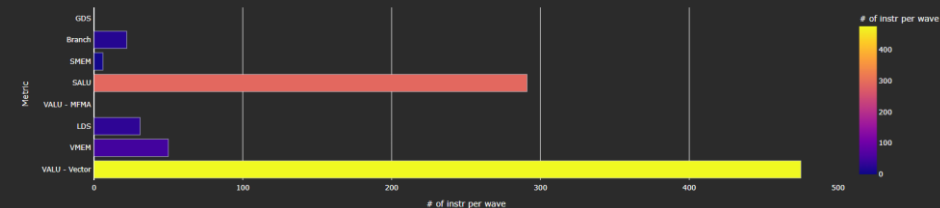


## 2. System Speed-of-Light

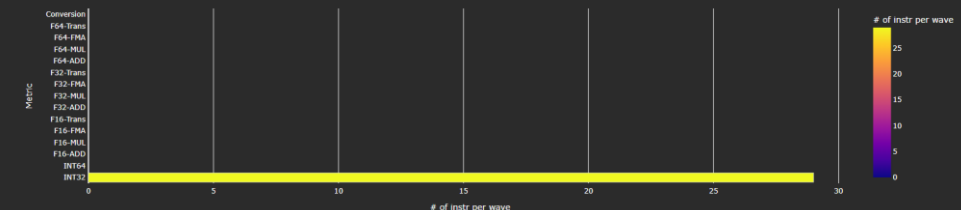
Metric	Value	Unit	Peak	Pop
VALU FLOPs	0.00	Gflop	23936.00	0.00
VALU IOPs	89.14	Gflop	23936.00	0.37
MFMA FLOPs (BF16)	0.00	Gflop	95744.00	0.00
MFMA FLOPs (F16)	0.00	Gflop	191488.00	0.00
MFMA FLOPs (F32)	0.00	Gflop	47872.00	0.00
MFMA FLOPs (F64)	0.00	Gflop	47872.00	0.00
MFMA IOPs (Int8)	0.00	Gflop	191488.00	0.00
Active CUs	58.00	Cus	110.00	52.73

## 10. Compute Units - Instruction Mix

### 10.1 Instruction Mix



### 10.2 VALU Arithmetic Instr Mix



# Omniperf analyze with Grafana™ GUI

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

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

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

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

A new directory will be created called workloads/vcopy\_all

Import the database to analyze in Grafana™ GUI:

```
$ omniperf database --import [connection options] -w workloads/vcopy_demo/mi200/  
ROC Profiler: /usr/bin/rocprof
```

```
-----  
Import Profiling Results  
-----
```

```
Pulling data from /root/test/workloads/vcopy_demo/mi200
```

```
The directory exists
```

```
Found sysinfo file
```

```
KernelName shortening enabled
```

```
Kernel name verbose level: 2
```

```
Password:
```

```
Password recieved
```

```
-- Conversion & Upload in Progress --
```

```
... ..
```

```
9 collections added.
```

```
Workload name uploaded
```

```
-- Complete! --
```

Metric	Speed of Light		Theoretical Max	Pct-of-Peak
	Avg	Unit		
VALU FLOPs	0	GFLOP	23,936	0%
VALU IOPs	379	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	75	CUs	110	68%
SALU Util	4	pct	100	4%
VALU Util	6	pct	100	6%
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%
Instr Cache BW	217	GB/s	6,093	4%
Scalar L1D Cache Hit Rate	100	pct	100	100%
Scalar L1D Cache BW	217	GB/s	6,093	4%
Vector L1D Cache Hit Rate	50	pct	100	50%
Vector L1D Cache BW	1,733	GB/s	11,968	14%
L2 Cache Hit Rate	36	pct	100	36%
L2-Fabric Read BW	434	GB/s	1,638	26%
L2-Fabric Write BW	301	GB/s	1,638	18%



---

# Key Insights from Omniparf Analyzer

---



# Grafana – System Info

General / Omnipperf\_v1.0.3\_pub ☆ ↻

Normalization "per Wave" ▾ Workload miperf\_aaa\_vcopy\_mi200 ▾ Dispatch Filter Enter variable value GCD 0 ▾ Kernels All ▾ Baseline Workload miperf\_asw\_vcopy\_mi200 ▾ Baseline Dispatch Filter Enter variable value Baseline GCD 0 ▾ Baseline Kernels All ▾ Comparison Panels System Info ▾ TopN 5 ▾

## System Info

System Info		
Metric	Current	Baseline
Date	Tue Jul 5 20:50:45 2022 (UTC)	Tue Jun 21 18:31:40 2022 (CDT)
Host Name	6fb5ce5e50da	node-bp126-014a
Host CPU	AMD Eng Sample: 100-000000248-08_35/21_N	AMD Eng Sample: 100-000000248-08_35/21_N
Host Distro	Ubuntu 20.04.4 LTS	Ubuntu 20.04.4 LTS
Host Kernel	5.9.1-amdsos-build32-1+	5.9.1-amdsos-build32-1+
ROCm Version	5.1.3-66	5.2.0-9768
GFX SoC	mi200	mi200
GFX ID	gfx90a	gfx90a
Total SEs	8	8
Total SQCs	56	56
Total CUs	110	110
SIMDs/CU	4	4
Max Wavefronts Occupancy Per CU	32	32
Max Workgroup Size	1,024	1,024
L1Cache per CU (KB)	16	16
L2Cache (KB)	8,192	8,192
L2Cache Channels	32	32
Sys Clock (Max) - MHz	1,700	1,700
Memory Clock (Max) - MHz	1,600	1,600
Sys Clock (Cur) - MHz	800	800
Memory Clock (Cur) - MHz	1,600	1,600
HBM Bandwidth - GB/s	1,638.4	1,638.4

When assessing performance difference between two workloads (current and baseline), it's good to know the differences between underlying systems.

# Initial assessment with kernel statistics

Initial Assessment

Instruction/data flow

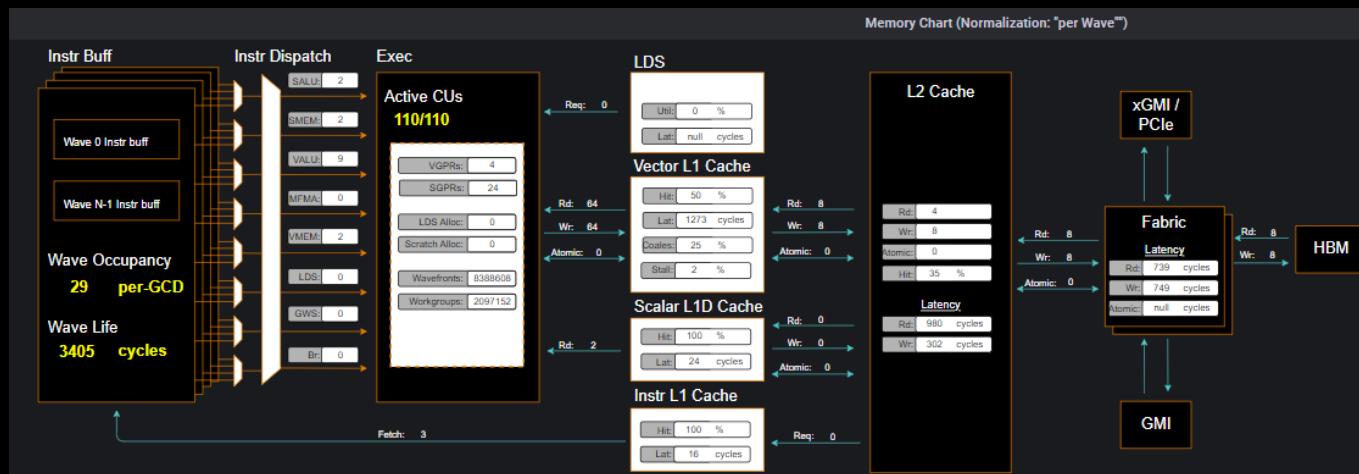
Speed-of-Light (SOL)

Omniperf tooling support

System SOL

Memory Chart

Kernel statistics



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

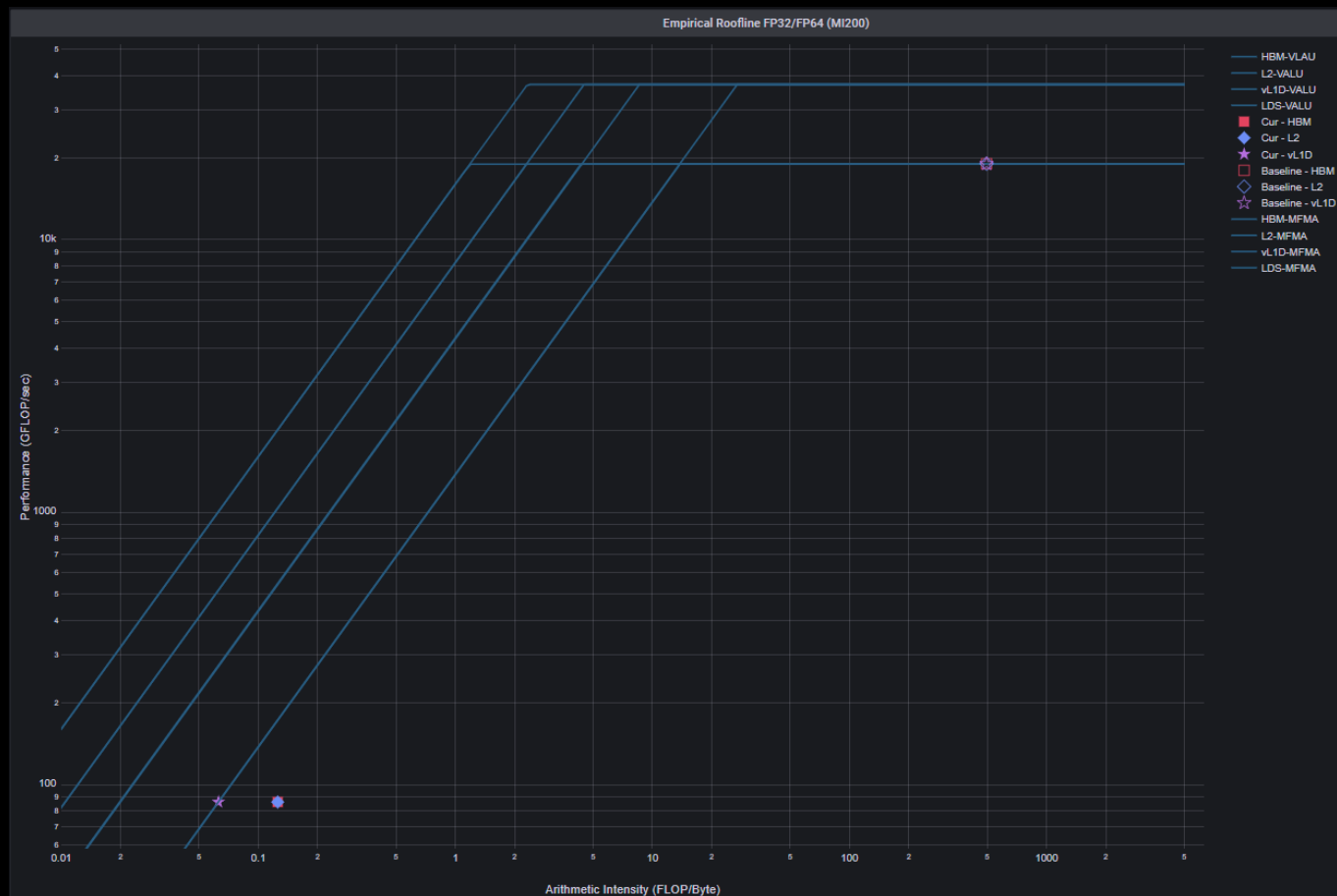
Performance margin

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



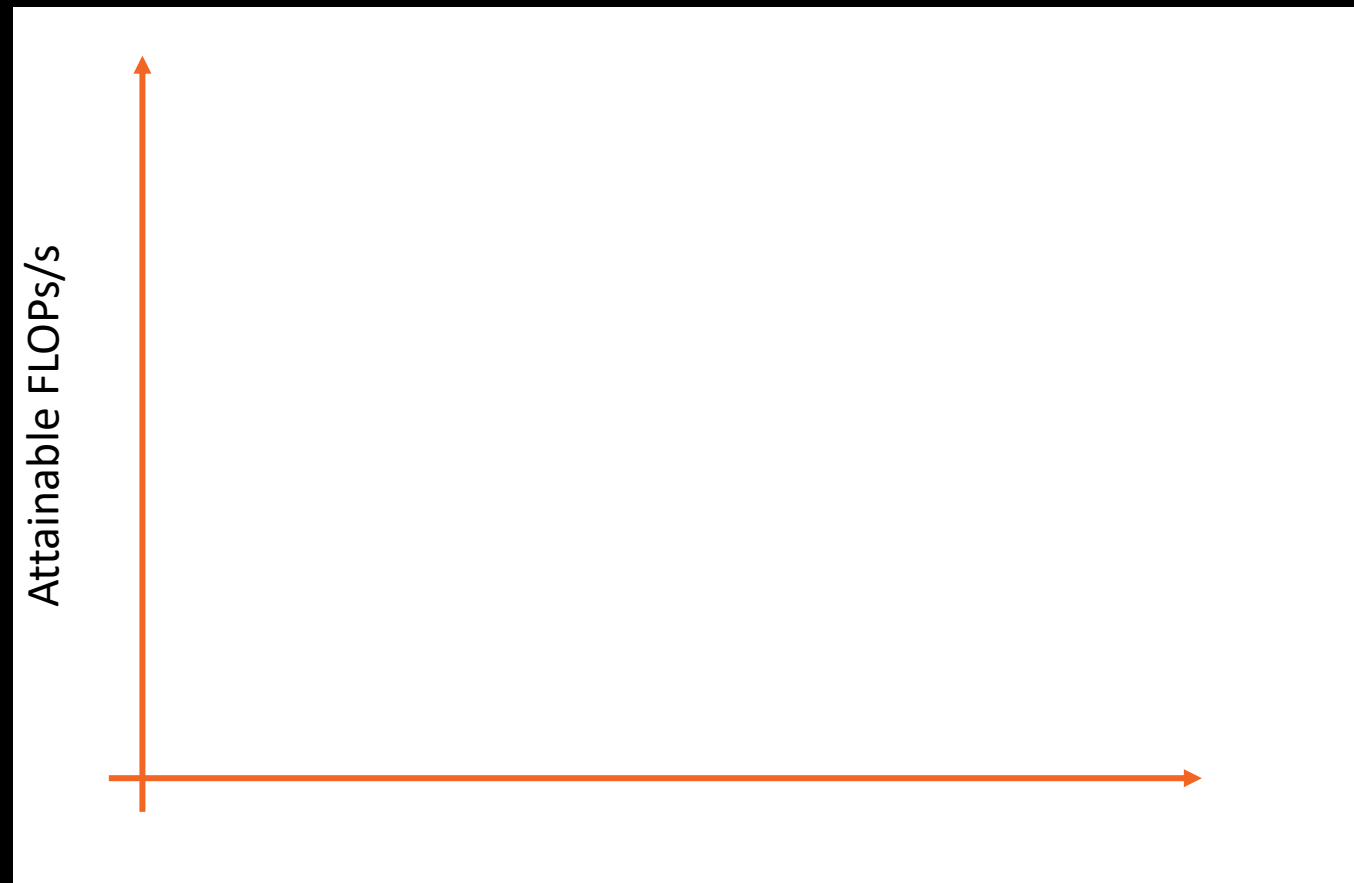
---

Background - What is a roofline?

---

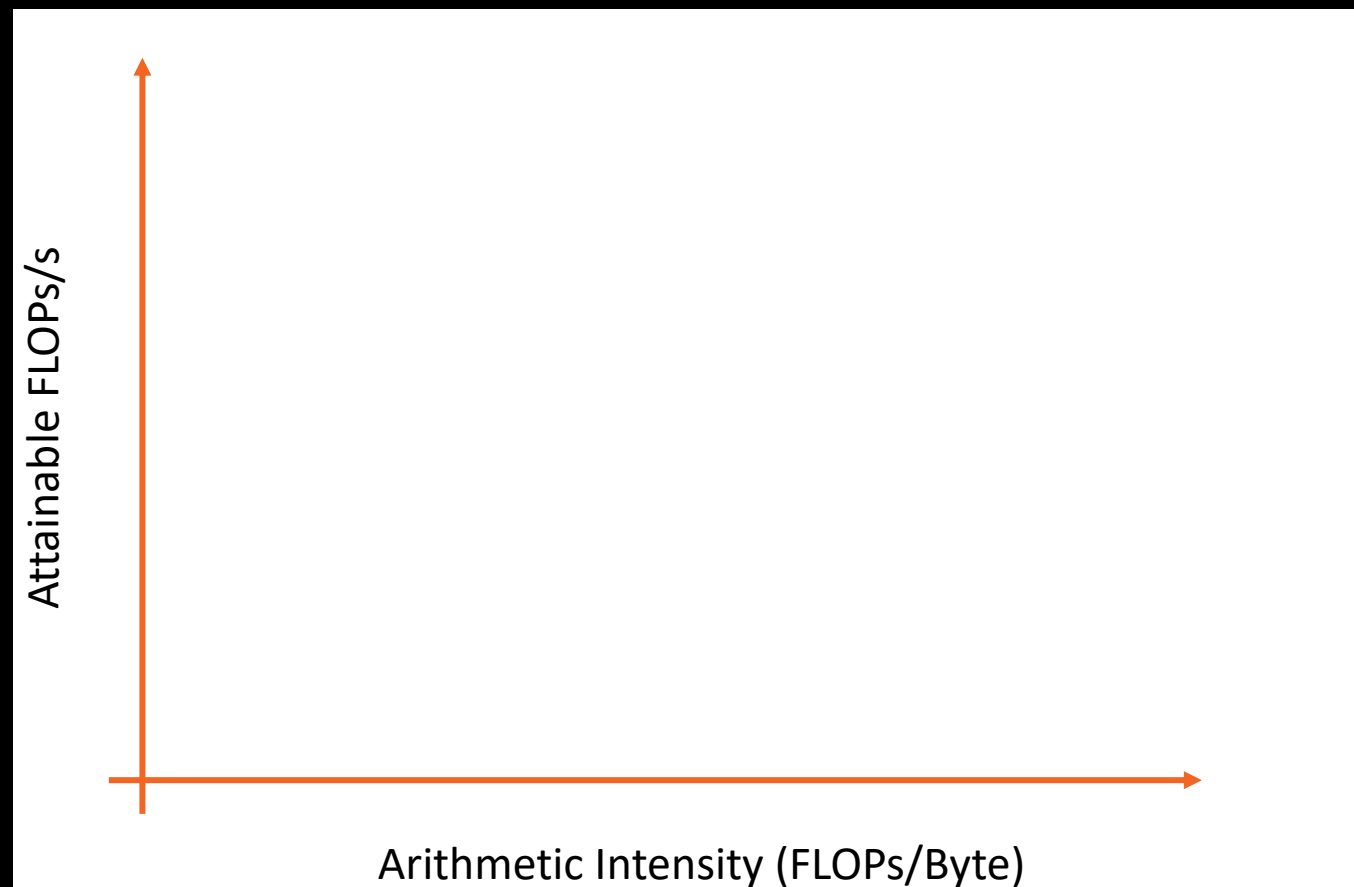
# Background – What is Roofline

- Attainable FLOPs/s
  - FLOPs/s rate as measured empirically on a given device
  - FLOP = floating point operation
  - FLOP counts for common operations
    - Add: 1 FLOP
    - Mul: 1 FLOP
    - FMA: 2 FLOP
  - FLOPs/s = Number of floating-point operations performed per second



# Background – What is Roofline

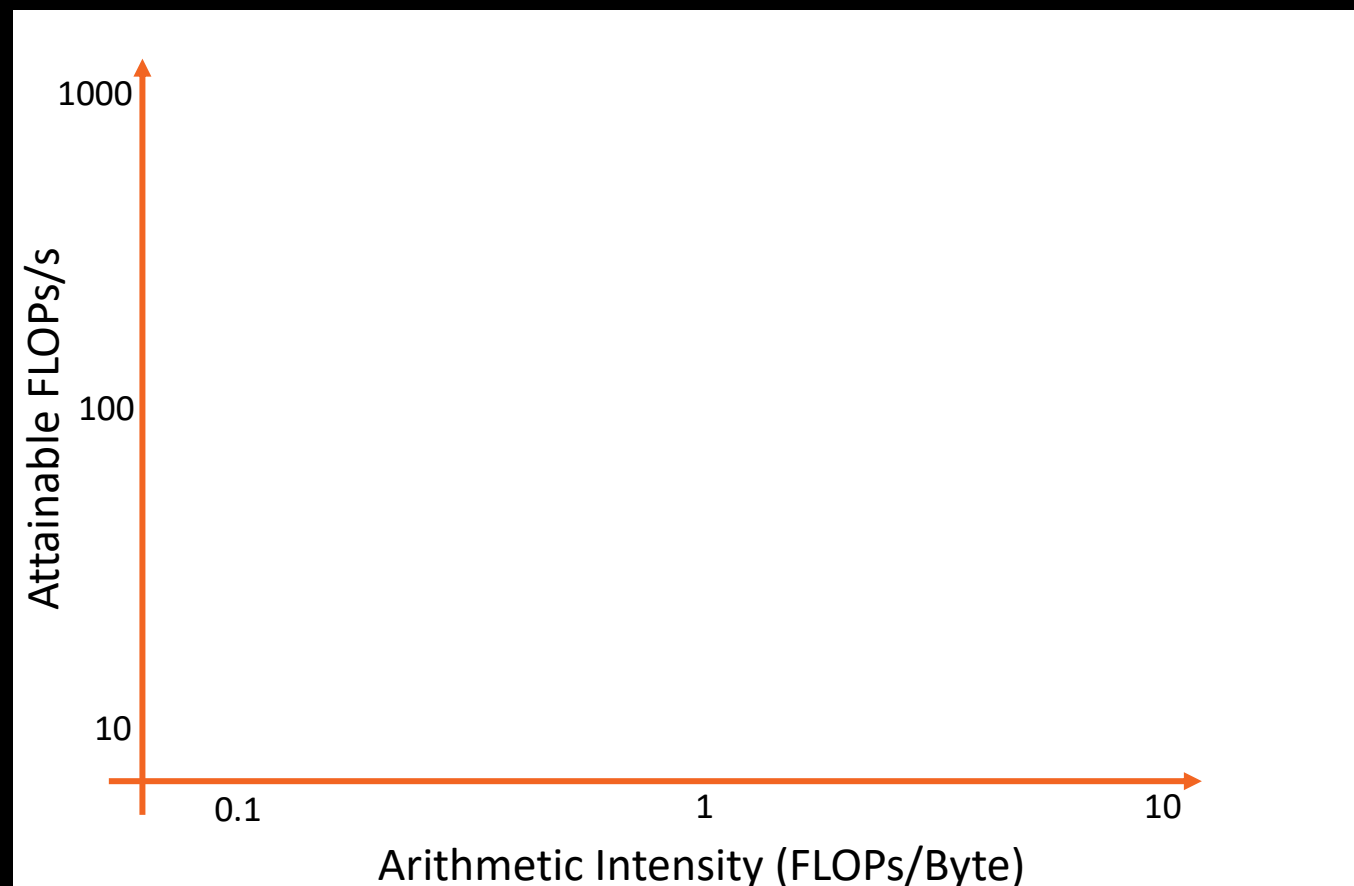
- Arithmetic Intensity (AI)
  - characteristic of the workload indicating how much compute (FLOPs) is performed per unit of data movement (Byte)
  - Ex:  $x[i] = y[i] + c$ 
    - FLOPs = 1
    - Bytes =  $1xRD + 1xWR = 4 + 4 = 8$
    - AI =  $1 / 8$



# Background – What is Roofline

- Log-Log plot

- makes it easy to doodle, extrapolate performance along Moore's Law, etc...



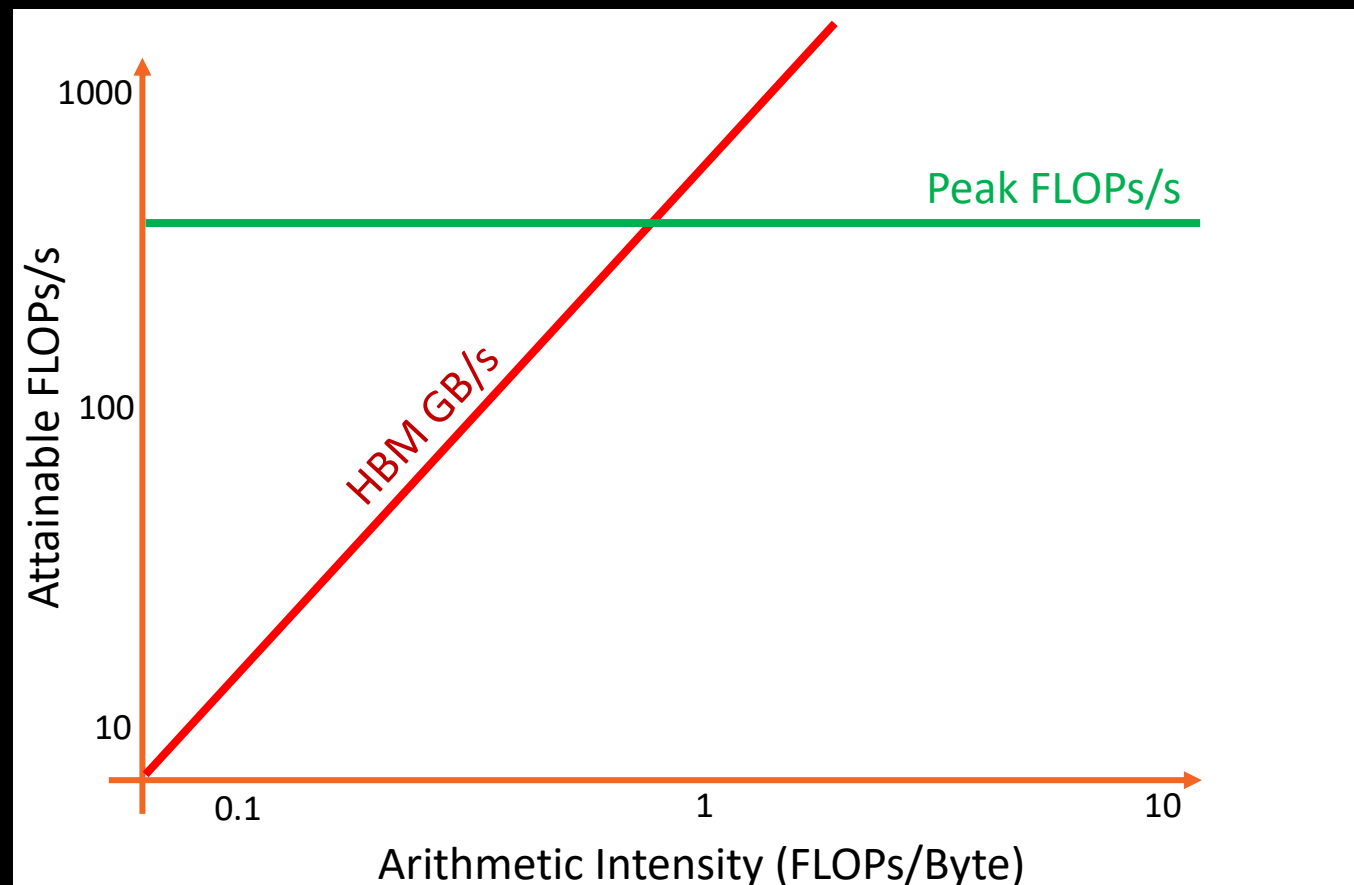
# Background – What is Roofline

## Roofline Limiters

- **Compute**
  - Peak FLOPs/s
- **Memory BW**
  - AI \* Peak GB/s

## Note:

- These are empirically measured values
- Different SKUs will have unique plots
- Individual devices within a SKU will have slightly different plots based on thermal solution, system power, etc.
- Omniperf uses suite of simple kernels to empirically derive these values
- These are NOT theoretical values indicating peak performance under “unicorn” conditions





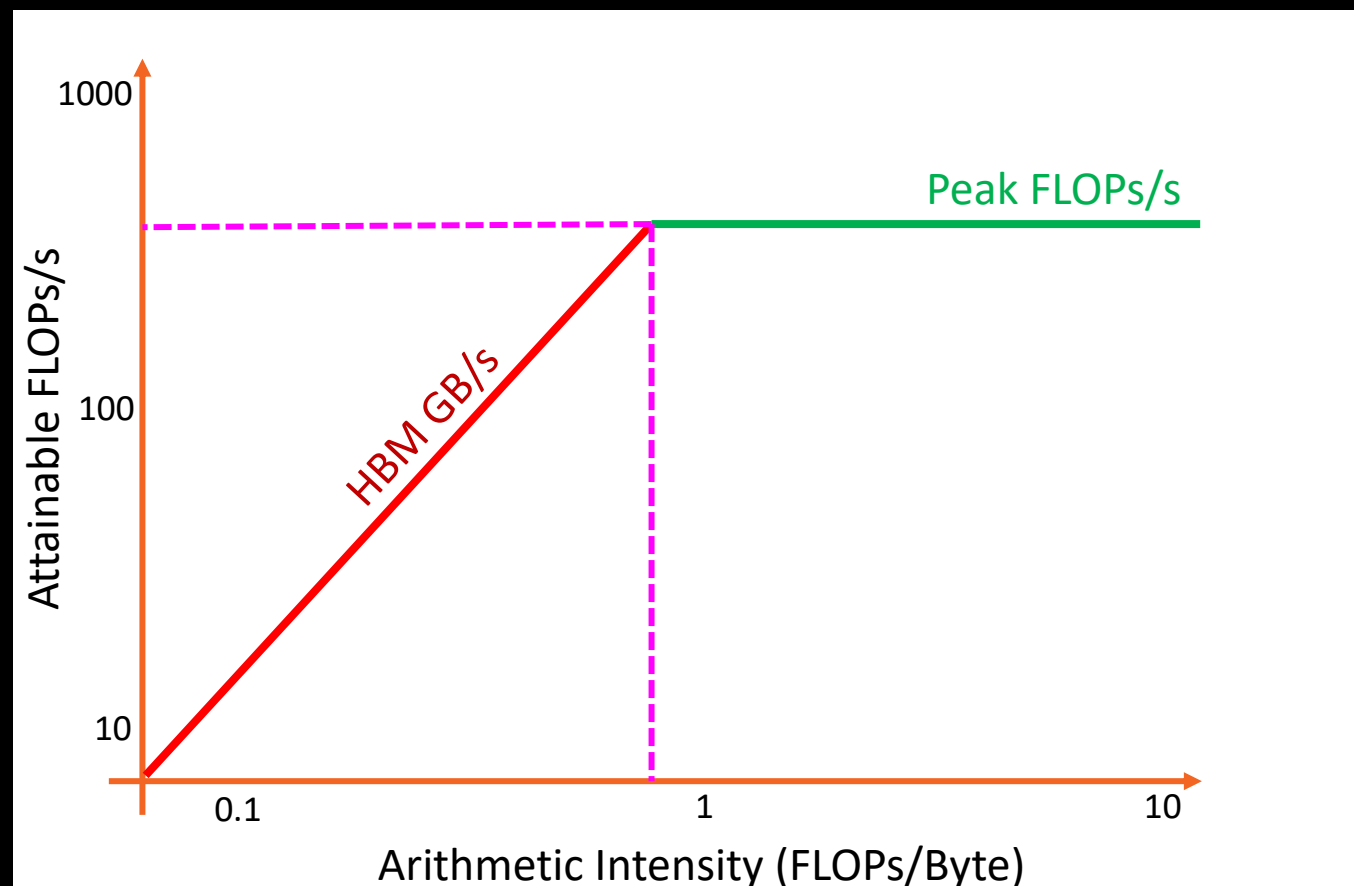
# Background – What is Roofline

• Attainable FLOPs/s =

$$\min \left\{ \begin{array}{l} \text{Peak FLOPs/s} \\ AI * \text{Peak GB/s} \end{array} \right.$$

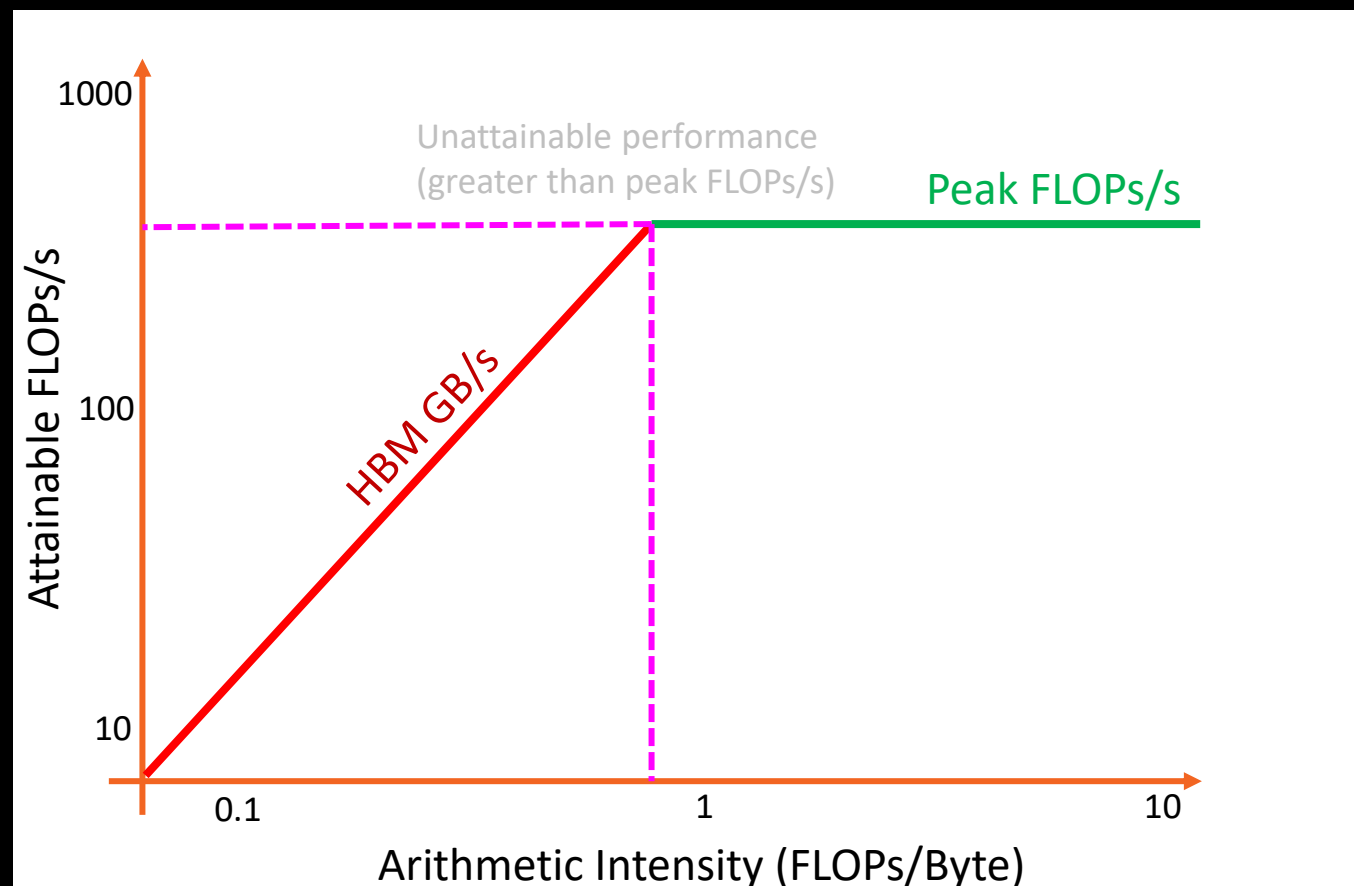
• Machine Balance:

- Where  $AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$
- Typical machine balance: 5-10 FLOPs/B
  - 40-80 FLOPs per double to exploit compute capability
- MI250x machine balance: ~16 FLOPs/B
  - 128 FLOPs per double to exploit compute capability



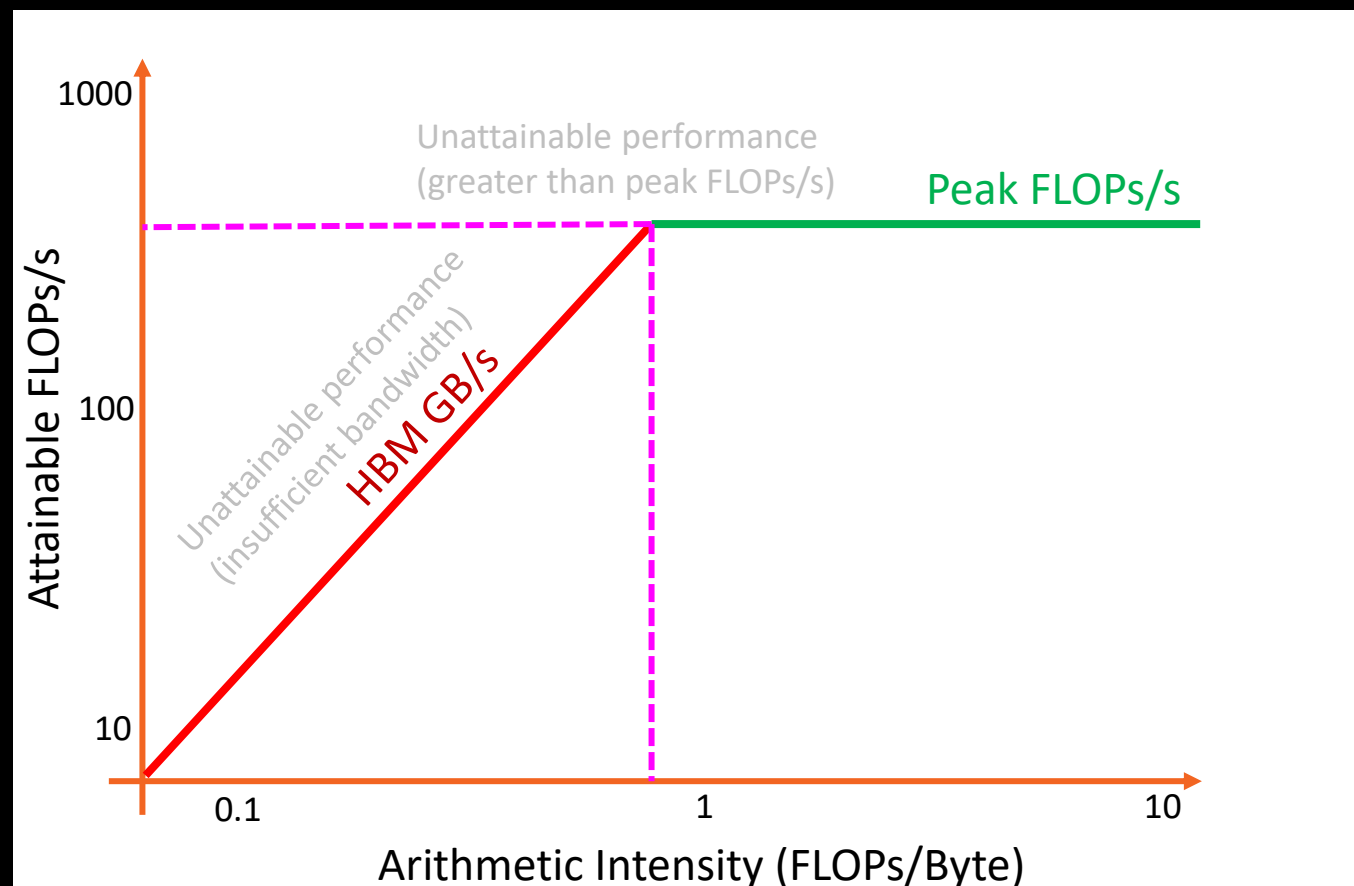
# Background – What is Roofline

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



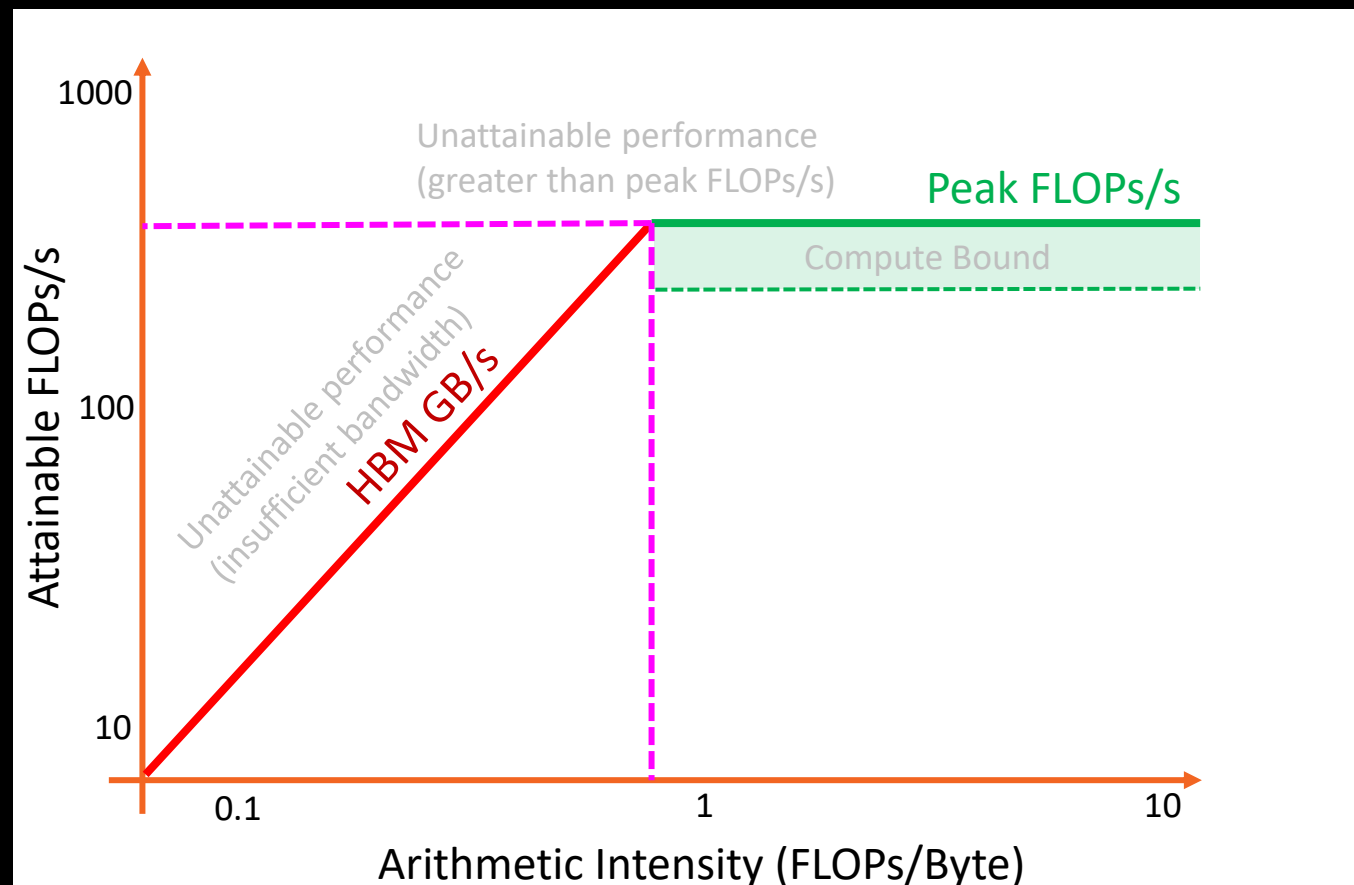
# Background – What is Roofline

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



# Background – What is Roofline

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



# Background – What is Roofline

• Attainable FLOPs/s =

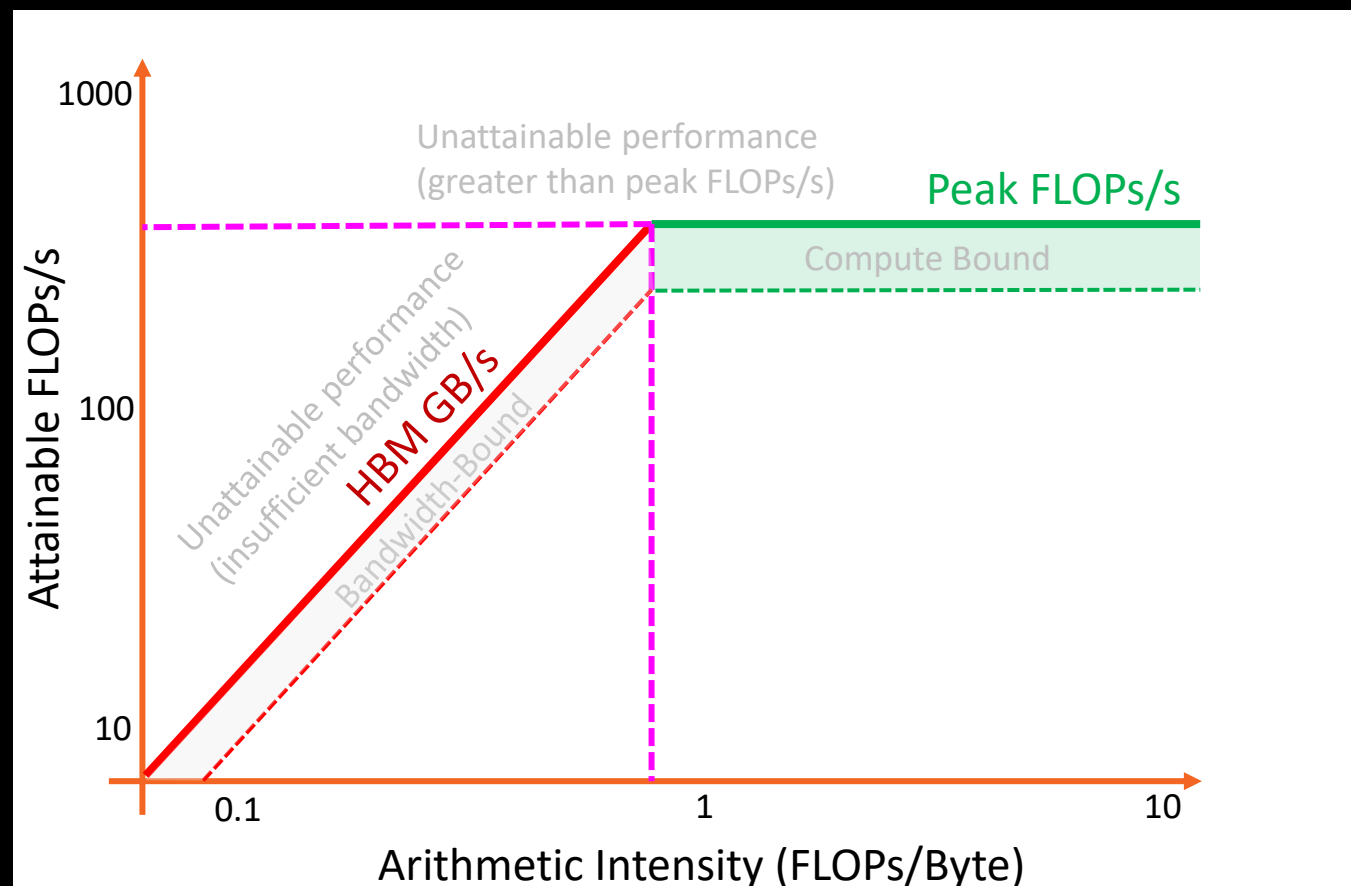
$$\min \left\{ \begin{array}{l} \text{Peak FLOPs/s} \\ AI * \text{Peak GB/s} \end{array} \right.$$

• Machine Balance:

$$\text{Where } AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$$

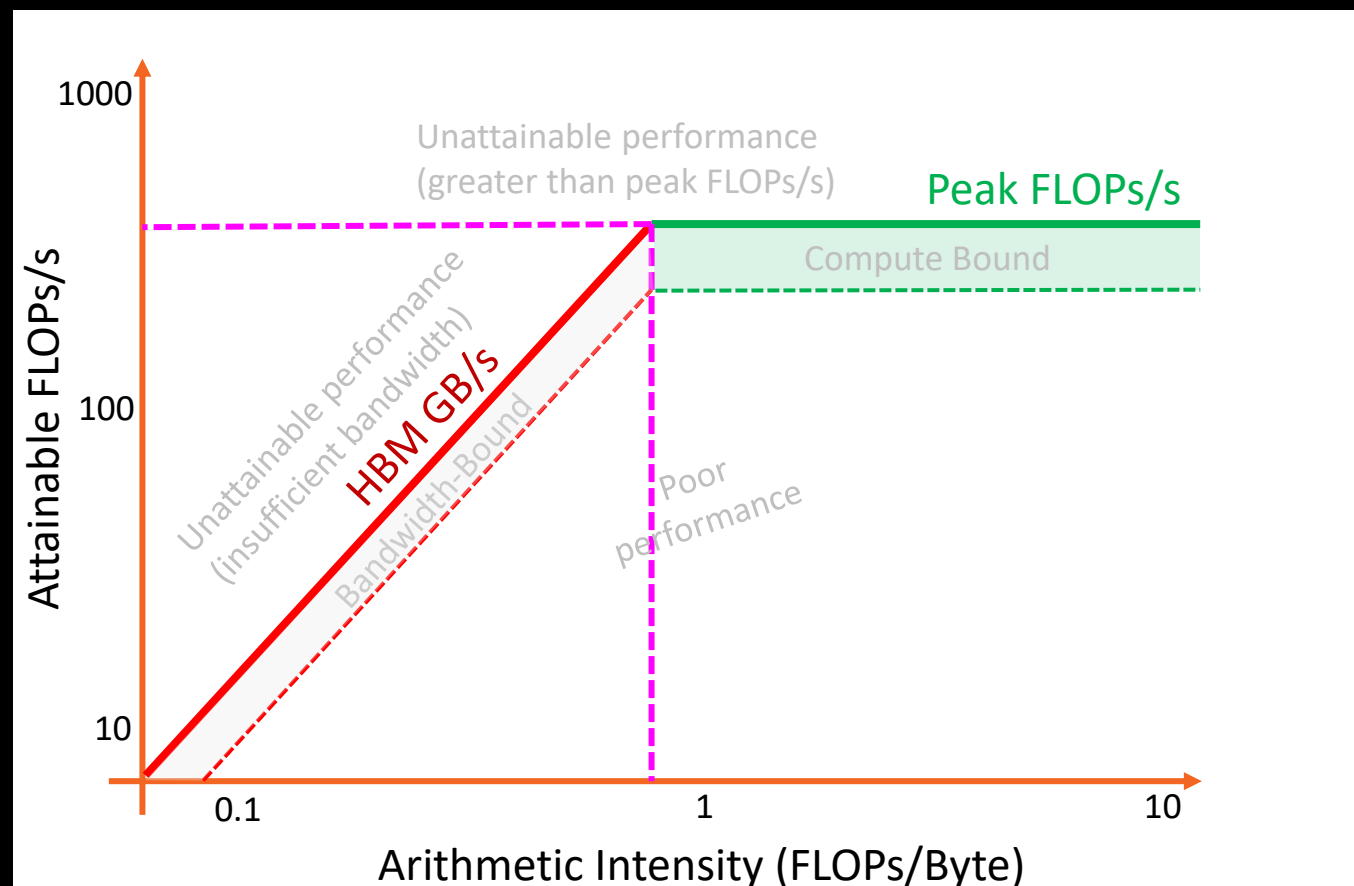
• Five Performance Regions:

- Unattainable Compute
- Unattainable Bandwidth
- Compute Bound
- Bandwidth Bound



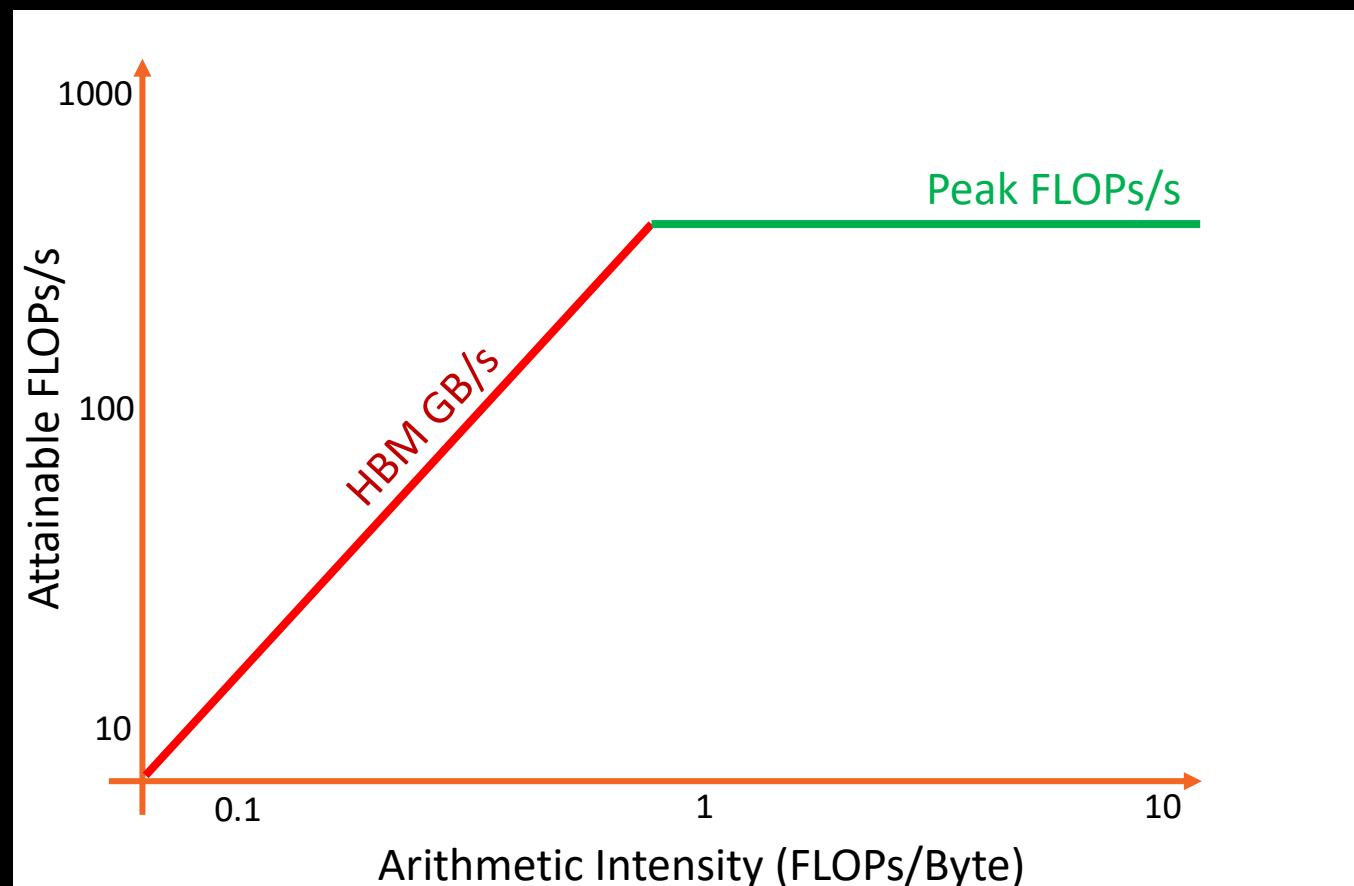
# Background – What is Roofline

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



# Background – What is Roofline

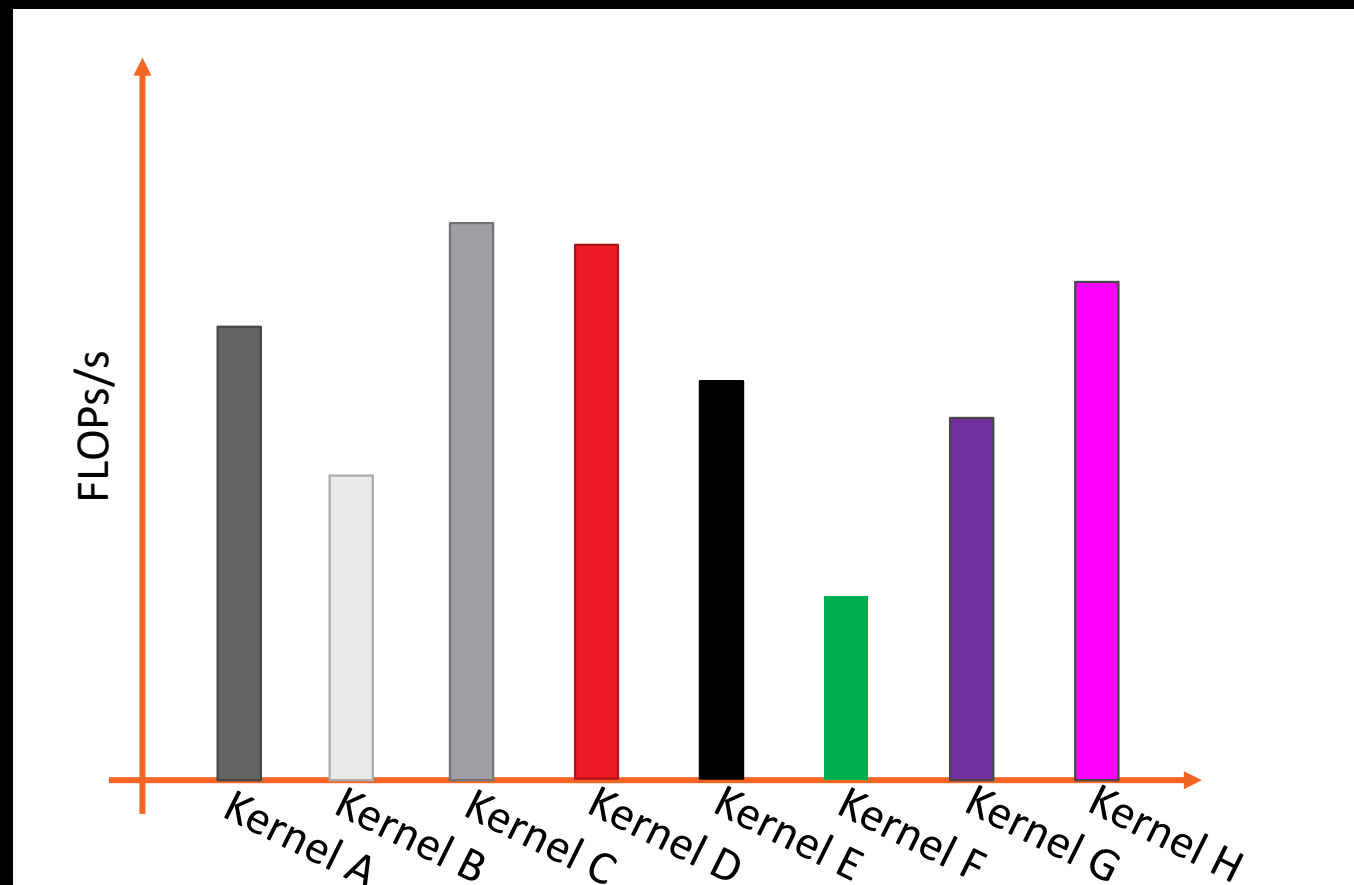
- Attainable FLOPs/s =
  - $\min \left\{ \begin{array}{l} \text{Peak FLOPs/s} \\ AI * \text{Peak GB/s} \end{array} \right.$
- Final result is a single roofline plot presenting the peak attainable performance (in terms of FLOPs/s) on a given device based on the arithmetic intensity of any potential workload
- We have an application independent way of measuring and comparing performance on any platform



# Background – What is “Good” Performance?

- Example:

- We run a number of kernels and measure FLOPs/s

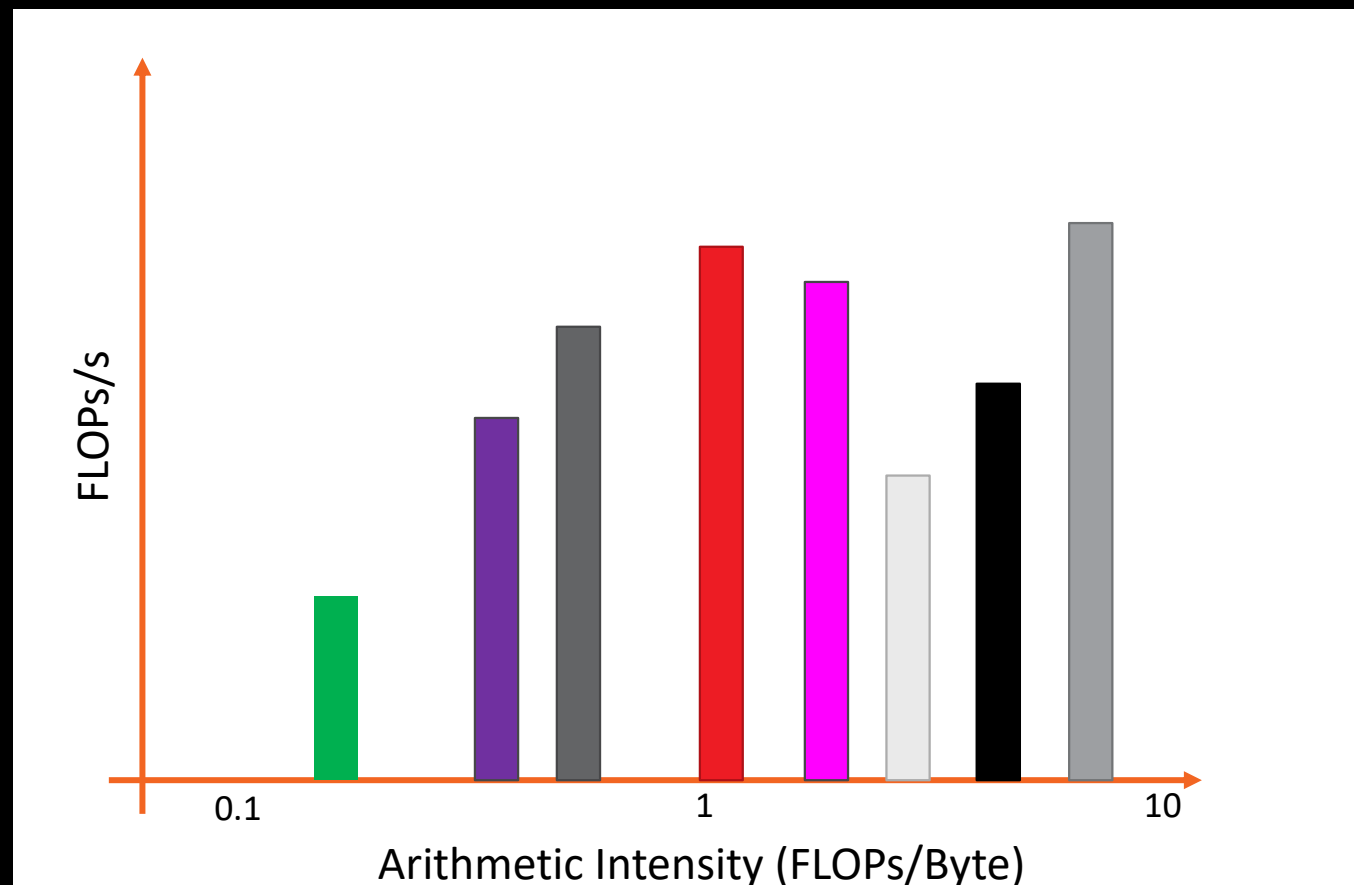




# Background – What is “Good” Performance?

- Example:

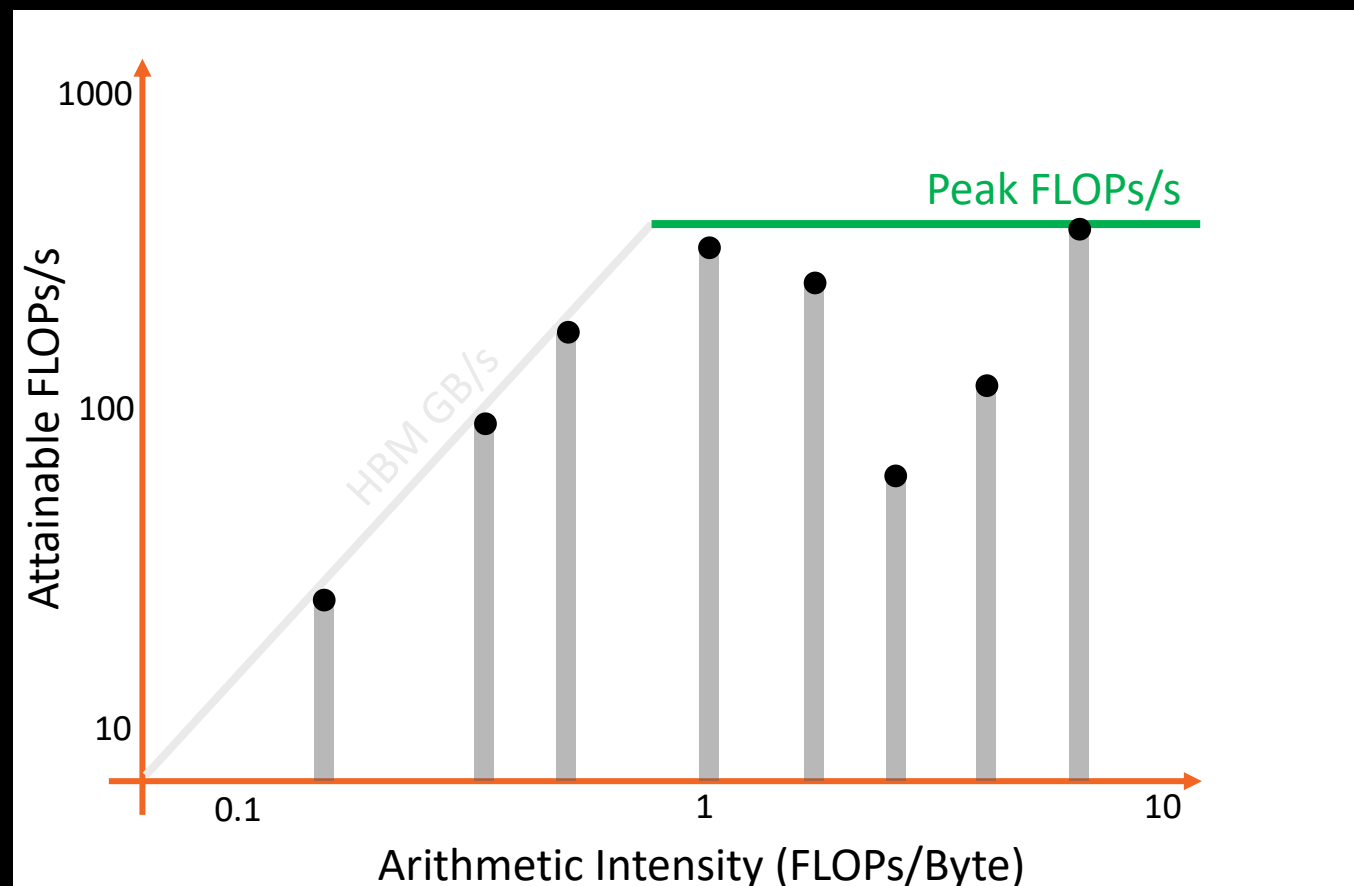
- We run a number of kernels and measure FLOPs/s
- Sort kernels by arithmetic intensity



# Background – What is “Good” Performance?

## Example:

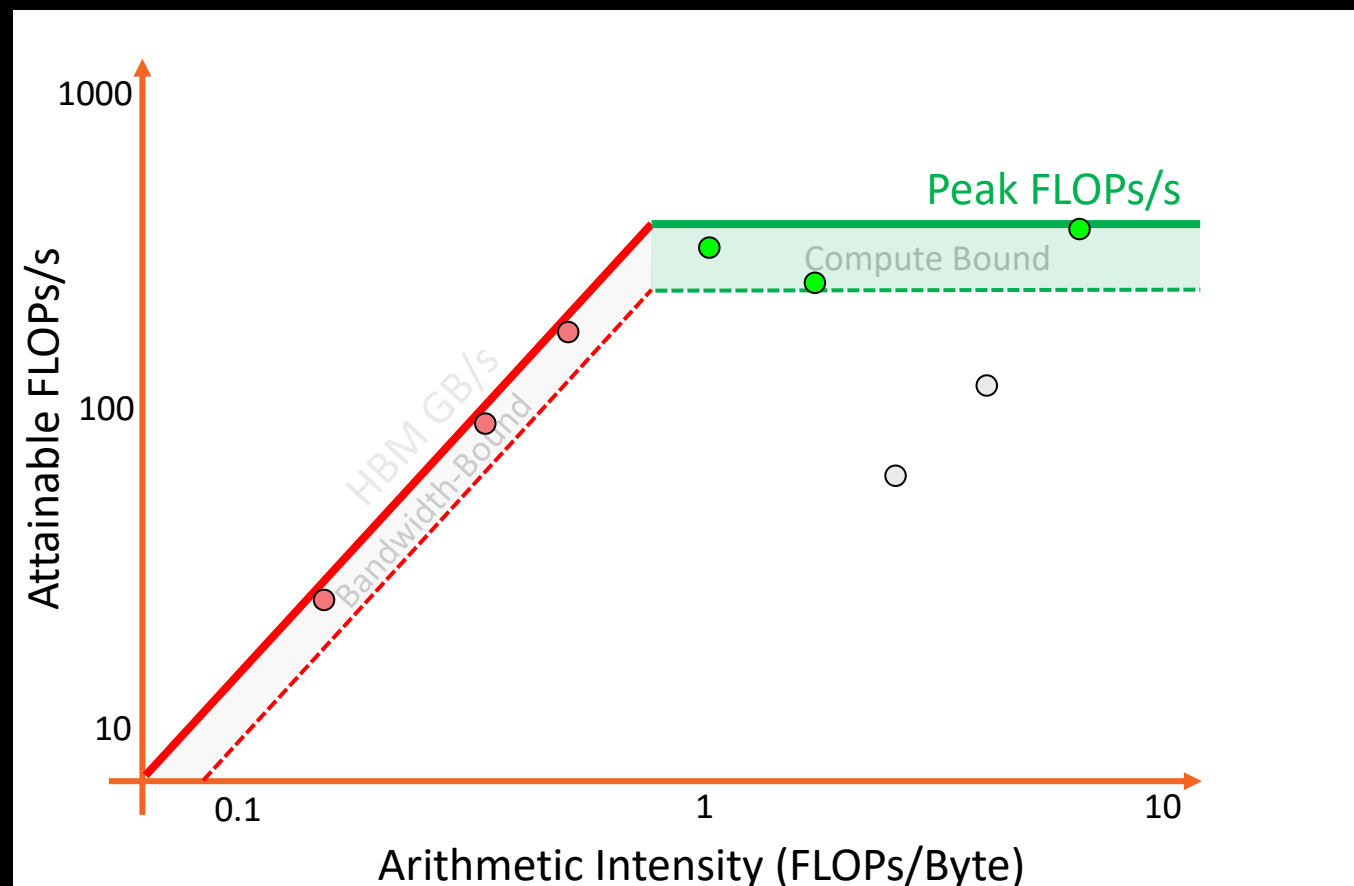
- We run a number of kernels and measure FLOPs/s
- Sort kernels by arithmetic intensity
- Compare performance relative to hardware capabilities



# Background – What is “Good” Performance?

## Example:

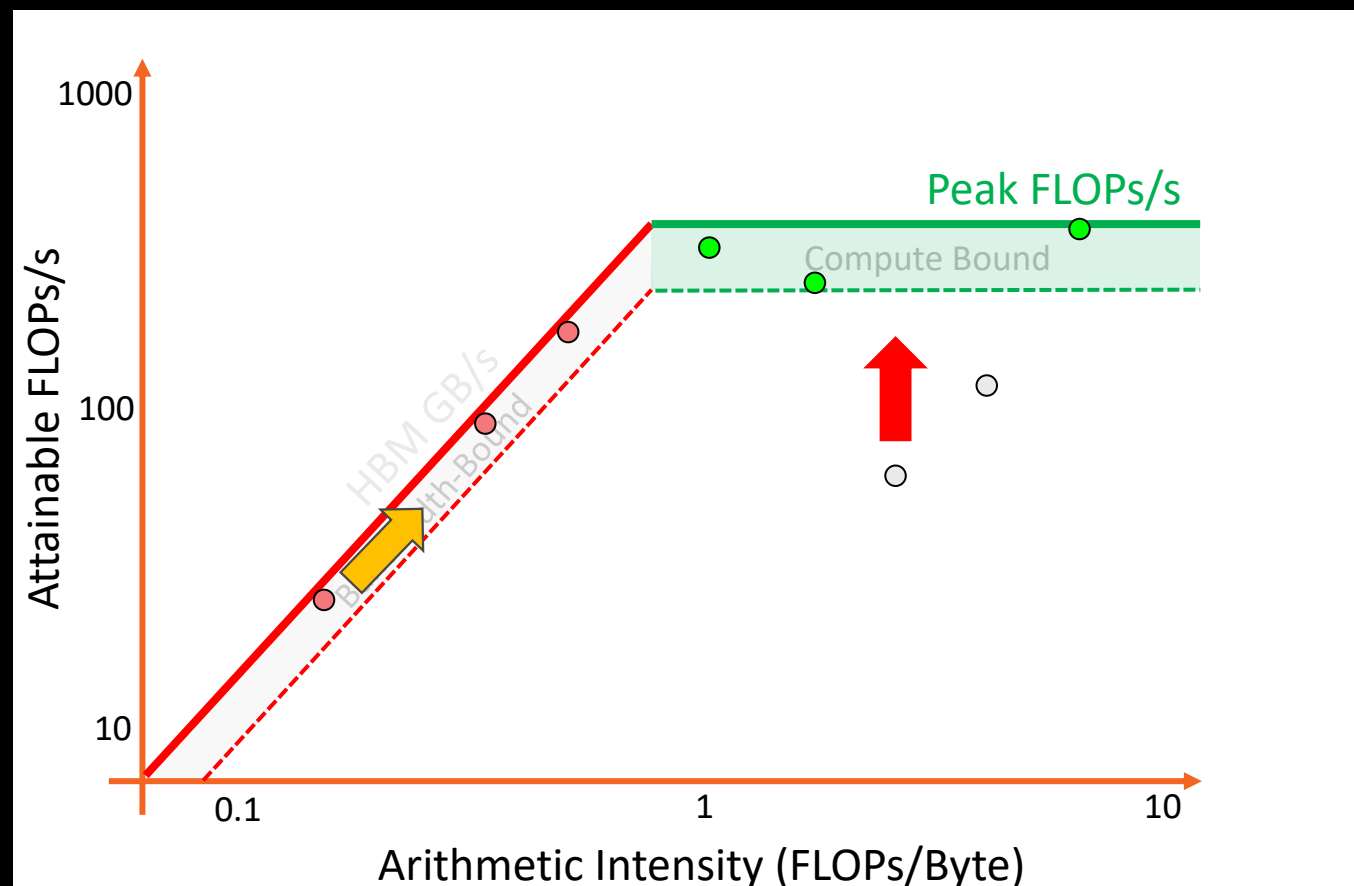
- We run a number of kernels and measure FLOPs/s
- Sort kernels by arithmetic intensity
- Compare performance relative to hardware capabilities
- Kernels near the roofline are making good use of computational resources
  - Kernels can have low performance (FLOPs/s), but make good use of BW



# Background – What is “Good” Performance?

## Example:

- We run a number of kernels and measure FLOPs/s
- Sort kernels by arithmetic intensity
- Compare performance relative to hardware capabilities
- Kernels near the roofline are making good use of computational resources
  - Kernels can have low performance (FLOPs/s), but make good use of BW
- Increase arithmetic intensity when bandwidth limited**
  - Reducing data movement increases AI
- Kernels not near the roofline *should*\* have optimizations that can be made to get closer to the roofline



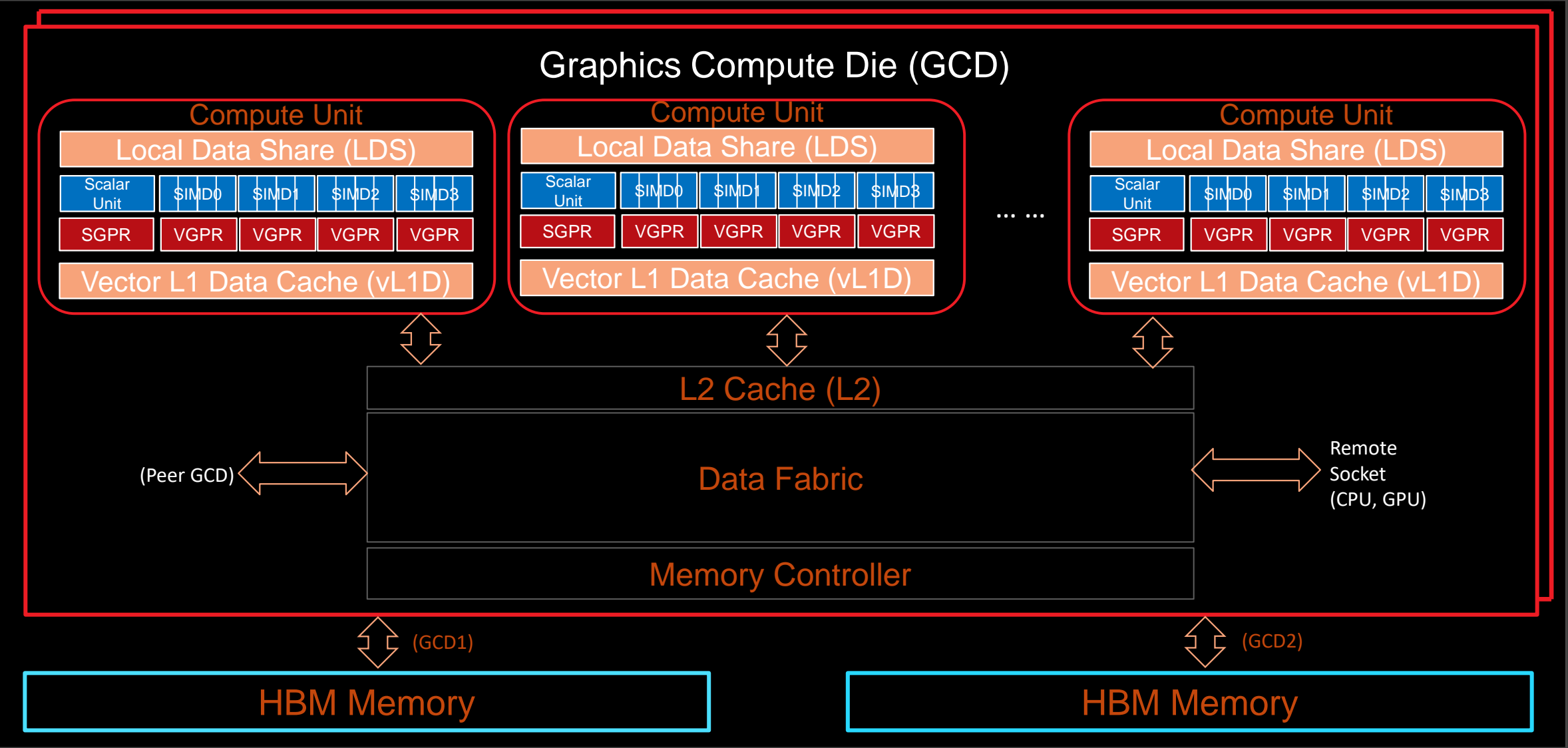
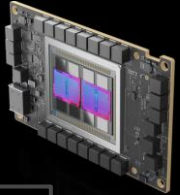


---

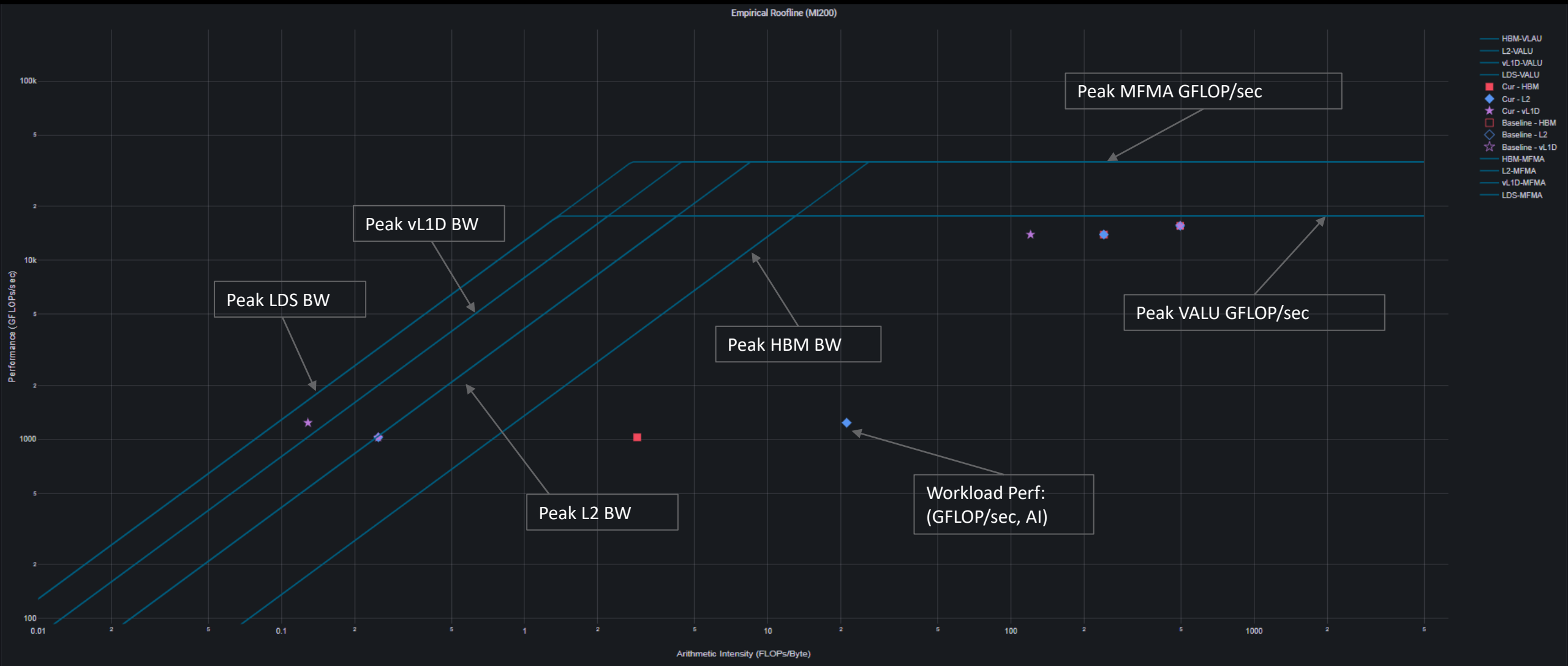
# Roofline Calculations on AMD Instinct™ MI200 GPUs

---

# Overview - AMD Instinct™ MI200 Architecture



# Empirical Hierarchical Roofline on MI200 - Overview



# Empirical Hierarchical Roofline on MI200 – Roofline Benchmarking

- Empirical Roofline Benchmarking
  - Measure achievable Peak FLOPS
    - VALU: F32, F64
    - MFMA: F16, BF16, F32, F64
  - Measure achievable Peak BW
    - LDS
    - Vector L1D Cache
    - L2 Cache
    - HBM
- Internally developed micro benchmark algorithms
  - Peak VALU FLOP: axpy
  - Peak MFMA FLOP: Matrix multiplication based on MFMA intrinsic
  - Peak LDS/vL1D/L2 BW: Pointer chasing
  - Peak HBM BW: Streaming copy

```

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

```



# Empirical Hierarchical Roofline on MI200 – Perfmon counters

- Weight
  - ADD: 1
  - MUL: 1
  - FMA: 2
  - Transcendental: 1
- FLOP Count
  - VALU: derived from VALU math instructions (assuming 64 active threads)
  - MFMA: count FLOP directly, in unit of 512
- Transcendental Instructions (7 in total)
  - $e^x$ ,  $\log(x)$  : F16, F32
  - $\frac{1}{x}$ ,  $\sqrt{x}$ ,  $\frac{1}{\sqrt{x}}$  : F16, F32, F64
  - $\sin x$ ,  $\cos x$  : F16, F32
- Profiling Overhead
  - Require 3 application replays

v\_rcp\_f64\_e32 v[4:5], v[2:3]  
 v\_sin\_f32\_e32 v2, v2  
 v\_cos\_f32\_e32 v2, v2  
 v\_rsq\_f64\_e32 v[6:7], v[2:3]  
 v\_sqrt\_f32\_e32 v3, v2  
 v\_log\_f32\_e32 v2, v2  
 v\_exp\_f32\_e32 v2, v2

ID	HW Counter	Category
1	SQ_INSTS_VALU_ADD_F16	FLOP counter
2	SQ_INSTS_VALU_MUL_F16	FLOP counter
3	SQ_INSTS_VALU_FMA_F16	FLOP counter
4	SQ_INSTS_VALU_TRANS_F16	FLOP counter
5	SQ_INSTS_VALU_ADD_F32	FLOP counter
6	SQ_INSTS_VALU_MUL_F32	FLOP counter
7	SQ_INSTS_VALU_FMA_F32	FLOP counter
8	SQ_INSTS_VALU_TRANS_F32	FLOP counter
9	SQ_INSTS_VALU_ADD_F64	FLOP counter
10	SQ_INSTS_VALU_MUL_F64	FLOP counter
11	SQ_INSTS_VALU_FMA_F64	FLOP counter
12	SQ_INSTS_VALU_TRANS_F64	FLOP counter
13	SQ_INSTS_VALU_INT32	IOP counter
14	SQ_INSTS_VALU_INT64	IOP counter
15	SQ_INSTS_VALU_MFMA_MOPS_I8	IOP counter

ID	HW Counter	Category
16	SQ_INSTS_VALU_MFMA_MOPS_F16	FLOP counter
17	SQ_INSTS_VALU_MFMA_MOPS_BF16	FLOP counter
18	SQ_INSTS_VALU_MFMA_MOPS_F32	FLOP counter
19	SQ_INSTS_VALU_MFMA_MOPS_F64	FLOP counter
20	SQ_LDS_IDX_ACTIVE	LDS Bandwidth
21	SQ_LDS_BANK_CONFLICT	LDS Bandwidth
22	TCP_TOTAL_CACHE_ACCESSES_sum	vL1D Bandwidth
23	TCP_TCC_WRITE_REQ_sum	L2 Bandwidth
24	TCP_TCC_ATOMIC_WITH_RET_REQ_sum	L2 Bandwidth
25	TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum	L2 Bandwidth
26	TCP_TCC_READ_REQ_sum	L2 Bandwidth
27	TCC_EA_RDREQ_sum	HBM Bandwidth
28	TCC_EA_RDREQ_32B_sum	HBM Bandwidth
29	TCC_EA_WRREQ_sum	HBM Bandwidth
30	TCC_EA_WRREQ_64B_sum	HBM Bandwidth

# Empirical Hierarchical Roofline on MI200 - Arithmetic

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

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

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

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

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

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

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

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

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

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



\* 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_ACSESSES_sum
```



---

# Omniperf Performance Analyzer (cont..)

---

# Subsystem performance analysis

Memory subsystems

L2 Cache

HBM access

LDS

vL1D

Omniperf tooling support

L2 Cache SOL

L2 fabric metrics

Per-channel statistics

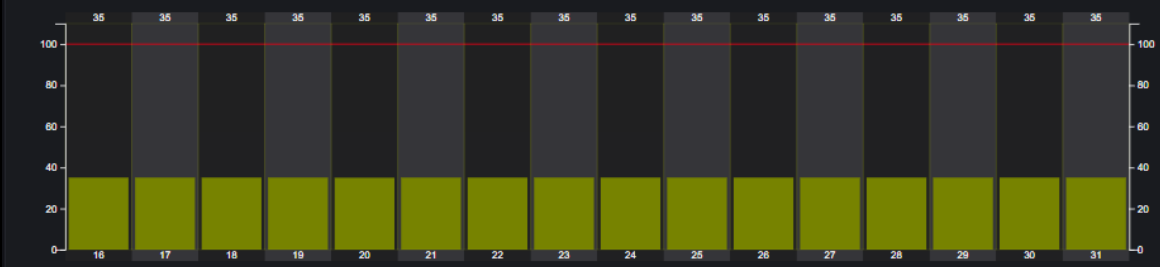
Speed-of-Light: L2 Cache



L2 - Fabric Transactions

Metric	Avg	Min	Max	Unit
Read BW	693,148,700,953	664,565,016,054	695,197,543,698	Bytes per Sec
Write BW	692,659,558,092	664,096,634,666	694,705,946,653	Bytes per Sec
Read (32B)	0	0	0	Req per Sec
Read (Uncached 32B)	2,304,240	1,434,649	2,370,898	Req per Sec
Read (64B)	10,830,448,452	10,383,828,376	10,862,461,620	Req per Sec
HBM Read	10,830,362,679	10,383,764,324	10,862,381,992	Req per Sec
Write (32B)	0	0	0	Req per Sec
Write (Uncached 32B)	0	0	0	Req per Sec
Write (64B)	10,822,805,595	10,376,509,917	10,854,780,416	Req per Sec
HBM Write	10,822,801,389	10,376,488,102	10,854,762,613	Req per Sec
Read Latency	739	732	801	Cycles
Write Latency	749	737	784	Cycles
Atomic Latency				Cycles
Read Stall	3	2	3	pct
Write Stall	6	5	8	pct

Cache Hit Rate % (Channel 16 - 31)



L2 - Fabric Interface Stalls (Cycles "per Wave")



# Shader compute components

Shader compute

Wavefront life

Instruction mix

Floating/  
Integer Ops

Compute  
pipeline

Instruction Mix



MFMA Arithmetic Instr Mix

MFMA Instr	Count
MFMA-i8	0
MFMA-F16	0
MFMA-BF16	0
MFMA-F32	0
MFMA-F64	995

Wavefront Runtime Stats

Metric	Avg	Min	Max	Unit
Kernel Time (Nanosec)	6,197,098	6,178,719	6,463,519	ns
Kernel Time (Cycles)	9,007,899	8,905,122	9,137,368	Cycle
Instr/wavefront	18	18	18	Instr/wavefro...
Wave Cycles	3,405	3,335	3,455	Cycles/wave
Dependency Wait Cycles	3,209	3,186	3,240	Cycles/wave
Issue Wait Cycles	165	112	193	Cycles/wave
Active Cycles	64	64	64	Cycles/wave
Wavefront Occupancy	3,198	3,166	3,210	Wavefronts

Speed-of-Light: Compute Pipeline



# Omniperf profile – Roofline only

Profile with roofline:

```
$ omniperf profile -n roofline_case_app --roof-only -- <CMD> <ARGS>
```

Analyze the profiled workload:

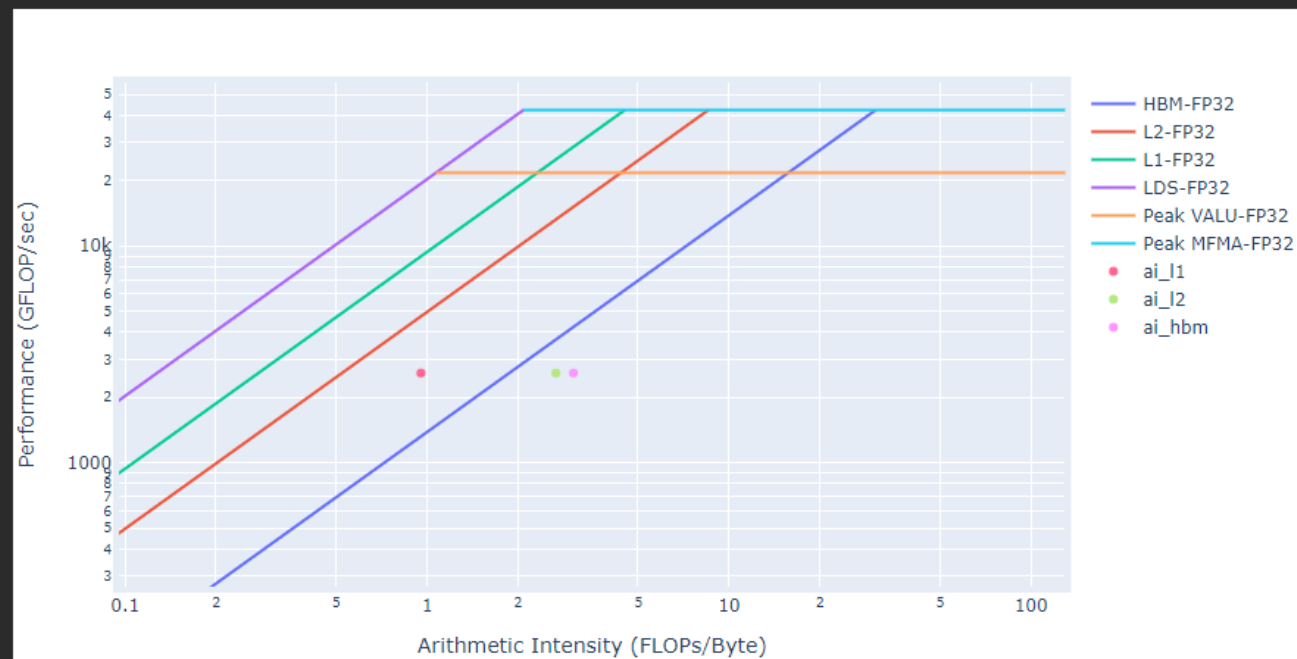
```
$ omniperf analyze -p path/to/workloads/roofline_case_app/mi200 --gui
```

Open web page <http://IP:8050/>

When profile with --roof-only, a PDF with the roofline will be created. In order to see the name of the kernels, add the --kernel-names and a second PDF will be created with names for the kernel markers:

```
$ omniperf profile -n roofline_case_app --roof-only --kernel-names -- <CMD> <ARGS>
```

## Empirical Roofline Analysis (FP32/FP64)

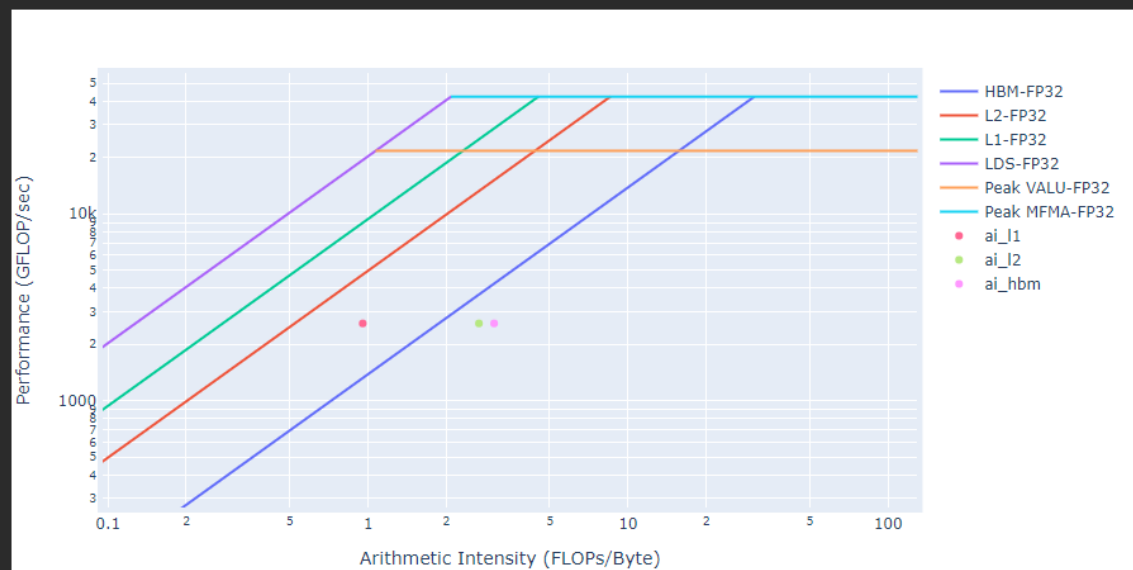


# Roofline Analysis – Kokkos code

Menu ▾ NORMALIZATION: per Wave ▾ KERNELS: Fetch: 346 GCD: ALL ▾ DISPATCH FILTER: ALL ▾ Report Bug

```
void
Kokkos::Experimental::Impl::hip_parallel_launch_constant_memory<Kokkos::Impl::ParallelFor<idfix_for<Hydro::HlIdMHD<2>
()::{lambda(int, int, int)#1}>(std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const&, int const&,
int const&, int const&, int const&, int const&, int const&, Hydro::HlIdMHD<2>()::{lambda(int, int, int)#1}>::lambda(int
const&)#1), Kokkos::RangePolicy<Kokkos::Experimental::HIP>, Kokkos::Experimental::HIP>, 256u, 1u>() [clone .kd]
```

## Empirical Roofline Analysis (FP32/FP64)



- Roofline: the first-step characterization of workload performance
  - Workload characterization
    - Compute bound
    - Memory bound
    - Performance margin
    - L1/L2 cache accesses
- Thorough SoC perf analysis for each subsystem to identify bottlenecks
  - HBM
  - L1/L2
  - LDS
  - Shader compute
  - Wavefront dispatch
- Omniperf tooling support
  - Roofline plot (float, integer)
  - Baseline roofline comparison
  - Kernel statistics



# SPI Resource Allocation

- Dispatch Bound
  - Wavefront dispatching failure due to resources limitation
    - Wavefront slots
    - VGPR
    - SGPR
    - LDS allocation
    - Barriers
    - Etc.
  - Omniperf tooling support
    - Shader Processor Input (SPI) metrics

## 6.2 SPI Resource Allocation

Metric	Avg	Min	Max	Unit
Wave request Failed (CS)	613303.00	613303.00	613303.00	Cycles
CS Stall	356961.00	356961.00	356961.00	Cycles
CS Stall Rate	62.95	62.95	62.95	Pct
Scratch Stall	0.00	0.00	0.00	Cycles
Insufficient SIMD Waveslots	0.00	0.00	0.00	Simd
Insufficient SIMD VGPRs	16252333.00	16252333.00	16252333.00	Simd
Insufficient SIMD SGPRs	0.00	0.00	0.00	Simd
Insufficient CU LDS	0.00	0.00	0.00	Cu
Insufficient CU Barries	0.00	0.00	0.00	Cu
Insufficient Bulky Resource	0.00	0.00	0.00	Cu
Reach CU Threadgroups Limit	0.00	0.00	0.00	Cycles
Reach CU Wave Limit	0.00	0.00	0.00	Cycles
VGPR Writes	4.00	4.00	4.00	Cycles/wave
SGPR Writes	5.00	5.00	5.00	Cycles/wave



---

What if Grafana and web GUI crashes when loading performance data?  
(real case)

---

# When profiling produces too large data...

- We had an application that the realistic case was dispatching 6.7 million calls to kernels
- Executing Omnipperf without any options, it would take up to 36 hours to finish while single non instrumented execution takes less than 1 hour.
- HW counters add overhead
- We had totally around 9 GB of profiling data from 1 MPI process
- Uploading the data to a Grafana server was crashing Grafana server and we had to reboot the service
- Using standalone GUI was never finishing loading the data
  
- Omnipperf profile has an option called `-k` where you define which specific kernel to profile. You can define the id 0-9 of the top 10 kernels.
- This creates profiling data **only** for the selected kernel
- This way you can split the profiling data to 10 executions, one per kernel:
  - You can use different resources to do the experiments in parallel (remember there can be performance variation between different GPUs)
  - You can visualize each kernel

Profile with roofline for a specific kernel:

```
$ srun -N 1 -n 1 --ntasks-per-node=1 --gpus=1 --hint=nomultithread omniperf profile -n kernel_roof  
-k kernel_name --roof-only -- ./binary args
```



---

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];
        }
}

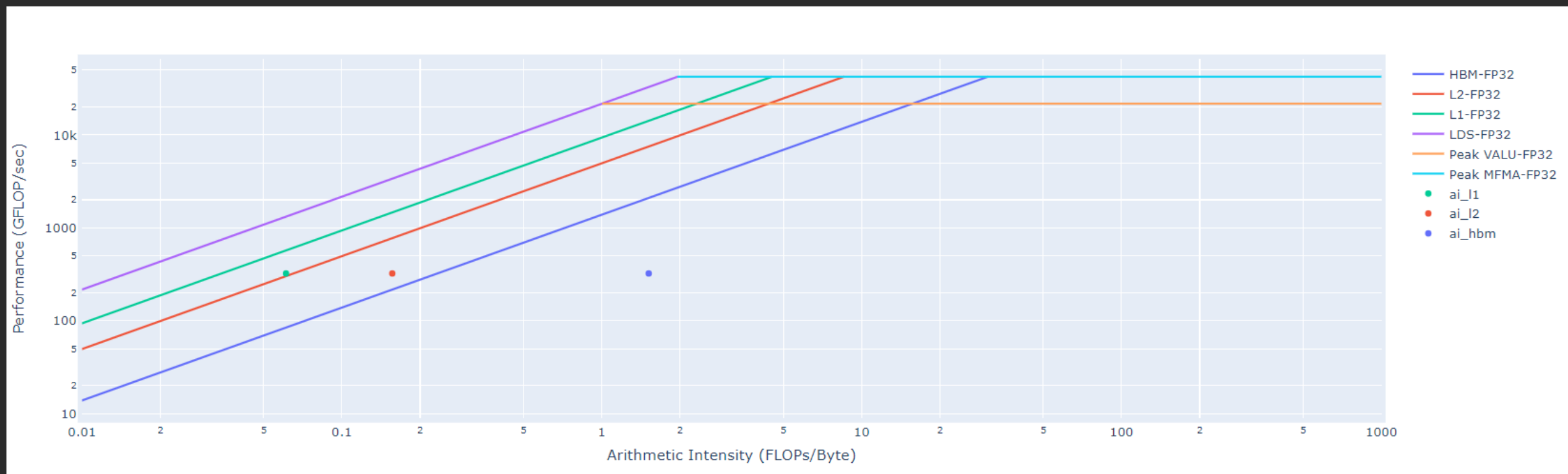
int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

    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

## Empirical Roofline Analysis (FP32/FP64)



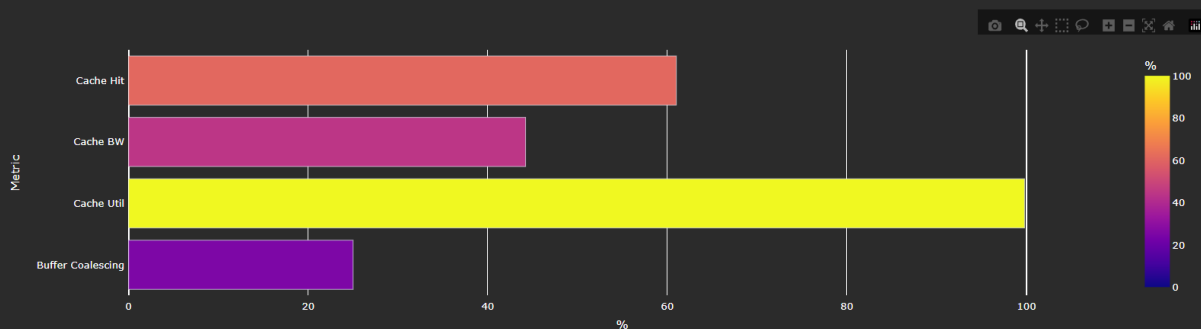
- Performance: almost 330 GFLOPs

# 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);

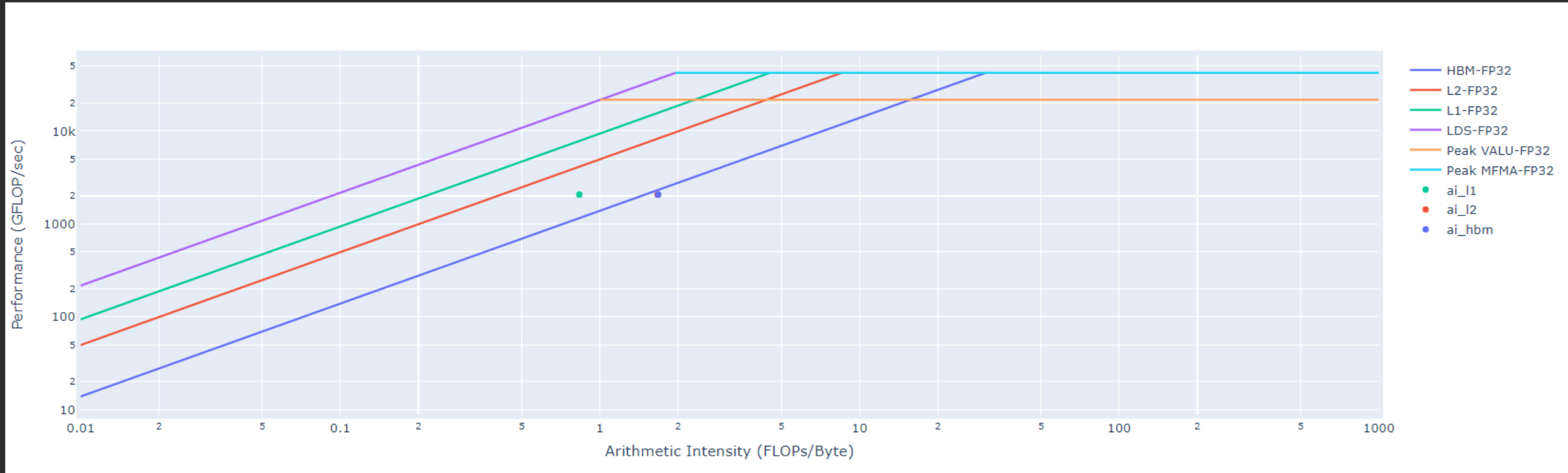
    int num_groups = (n+255)/256;
    int group_size = 256;
    daxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
    hipDeviceSynchronize();
}

```



# Roofline - Optimized

## Empirical Roofline Analysis (FP32/FP64)



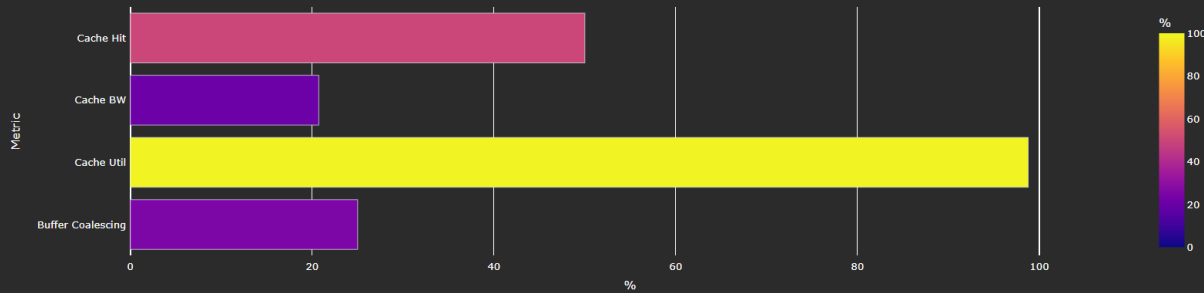
• Performance: almost 2 TFLOPs

# 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

# Questions?

# DISCLAIMERS AND ATTRIBUTIONS

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

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

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

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

**AMD** 