Hierarchical Roofline on AMD Instinct™ MI200 GPUs

ENCCS Workshop '22

Xiaomin Lu, Noah Wolfe

November 30, 2022

Agenda

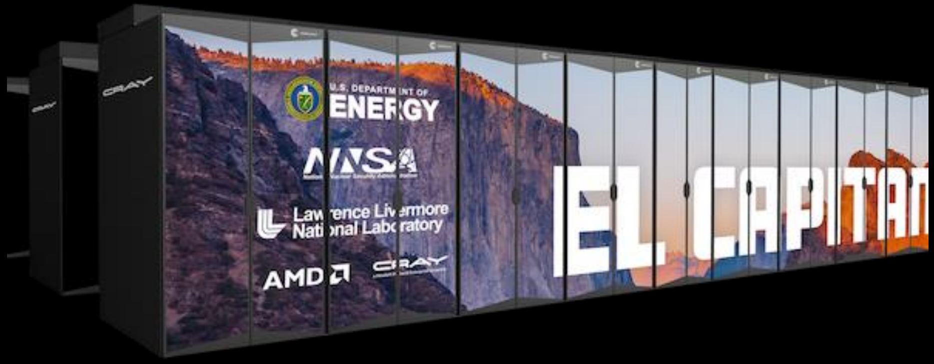
- Introduction
- Roofline Fundamentals
- Empirical Hierarchical Roofline on MI200
  - Overview
  - Roofline Arithmetic
  - Empirical Roofline Benchmarking
- Omniperf: Integrated Performance Analyzer for AMD GPUs
  - Architecture
  - Installation
  - Hello world
- Roofline Based Performance Analysis
  - Roofline characterization
  - SoC Performance and Bottleneck Analysis
- Examples
  - Add/Mul/FMA
  - N-Body
- HPC Application Results

AMD Fueling the Era of Exascale

EURO HPC/CSC
LUMI

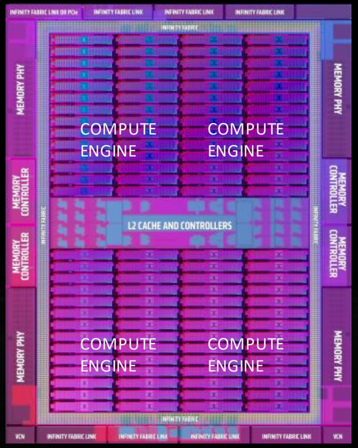
AMD
EPYC | INSTINCT

LAWRENCE LIVERMORE
EL CAPITAN



AMD INSTINCT™ MI250X ACCELERATOR

- TSMC 6NM TECHNOLOGY
- UP TO 110 CU PER GRAPHICS COMPUTE DIE
- 4 MATRIX CORES PER COMPUTE UNIT
- MATRIX CORES ENHANCED FOR HPC
- 8 INFINITY FABRIC LINKS PER DIE
- SPECIAL FP32 OPS FOR DOUBLE THROUGHPUT

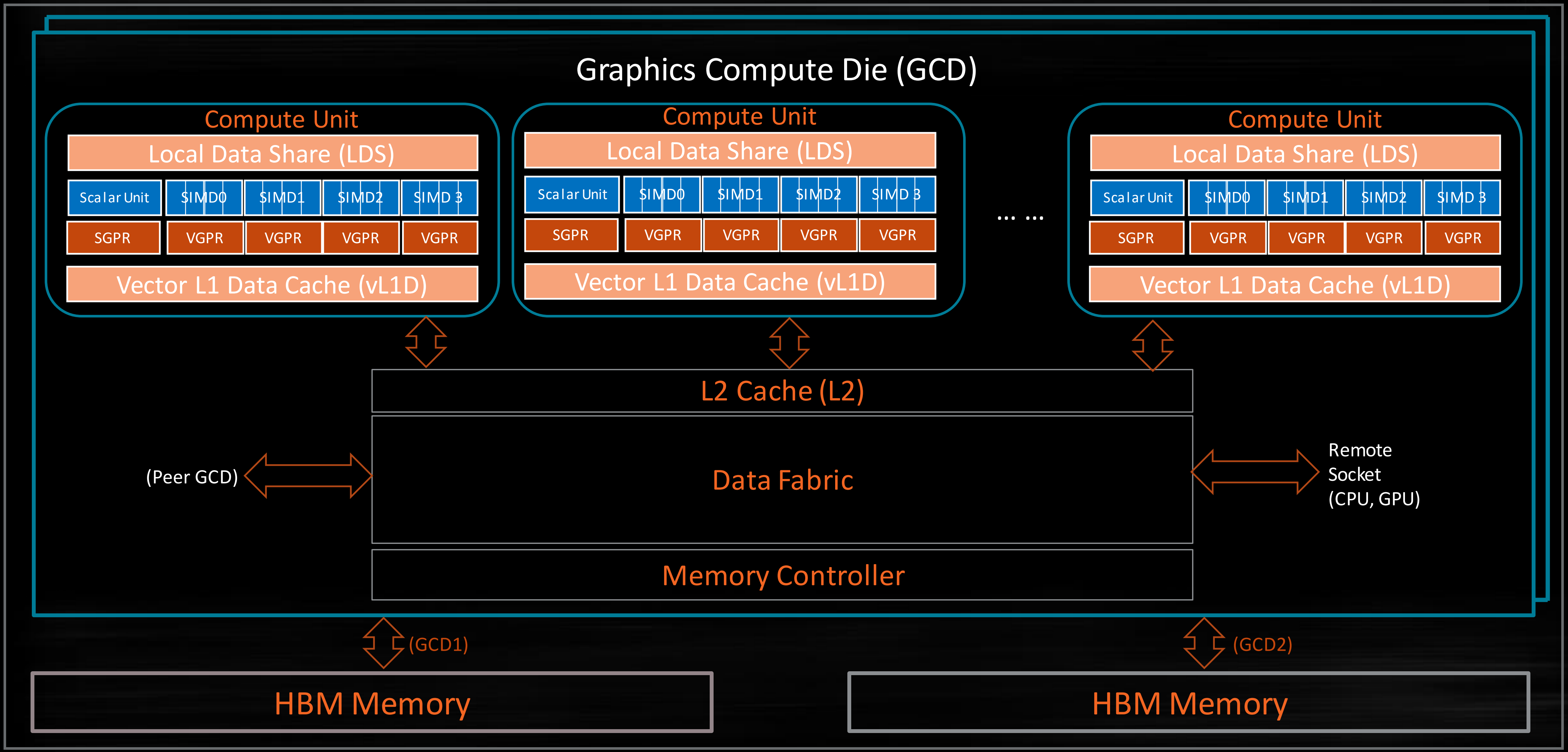


FRONTIER NODE AT A GLANCE

- Optimized 3rd Gen AMD EPYC™ processor
- Four Instinct™ MI250X accelerators
- Coherent connectivity
  - Via Infinity Fabric™ interconnect
  - Tightly integrated
  - Unified memory space

AMD

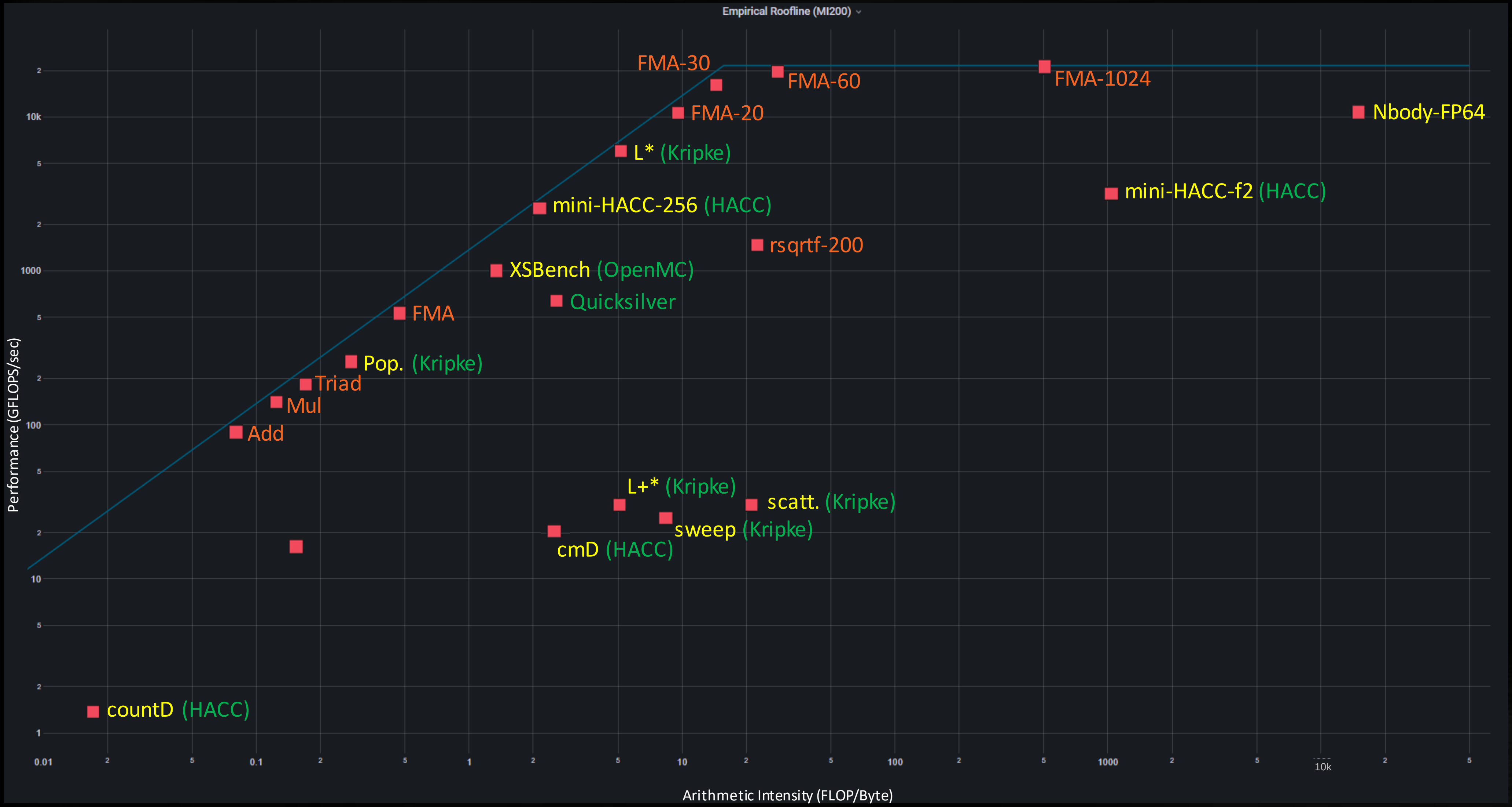
Overview - AMD Instinct™ MI200 Architecture



[Public]

Roofline – All Workloads

Orange: Synthetic Workload Yellow: Proxy app Green: Full app



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-62

Background – What is Roofline

Background – What is Roofline

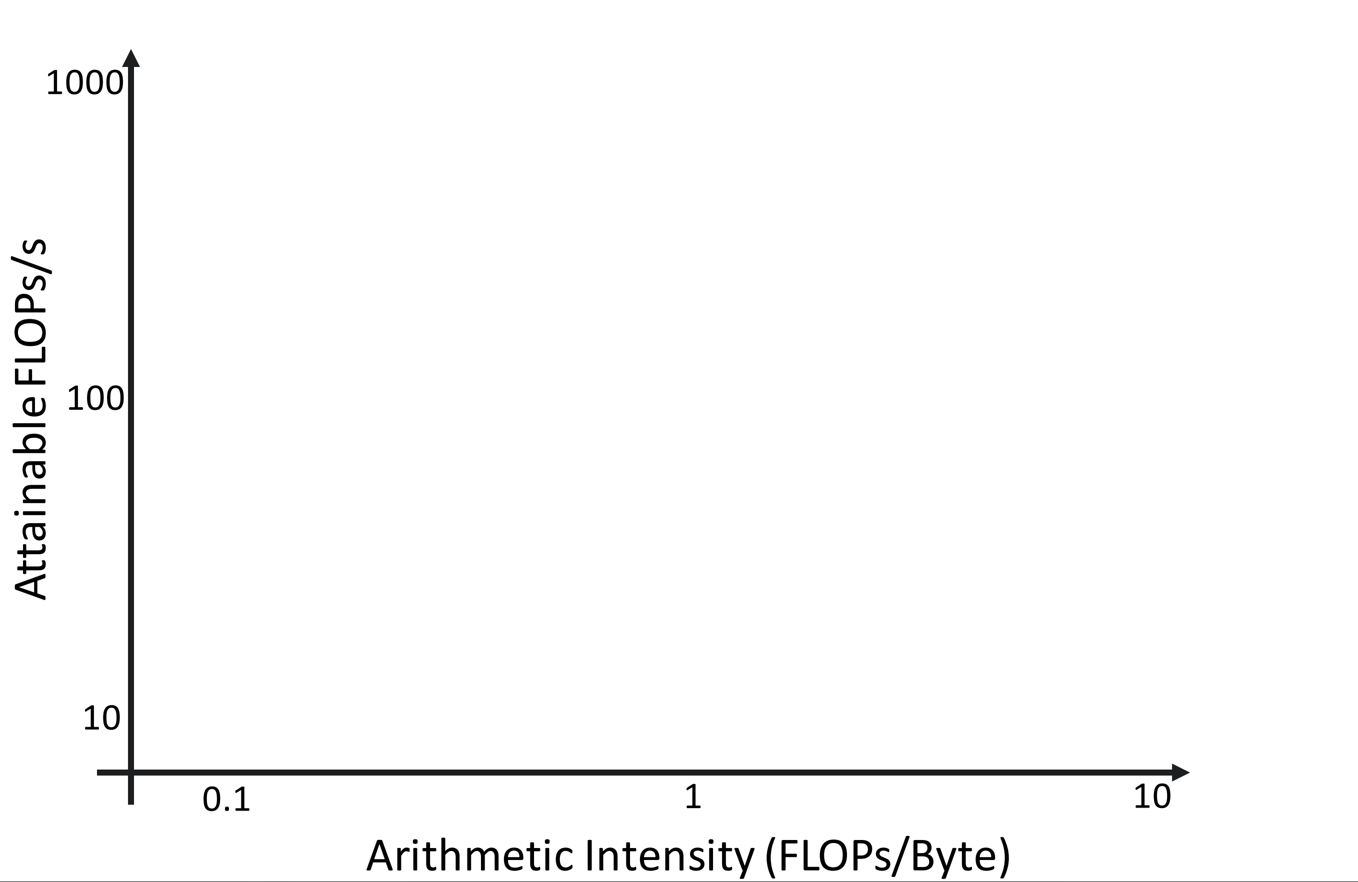
- Attainable FLOPs/s
  - FLOPs/s rate as measured empirically on a given device
  - FLOP = floating point operation
  - FLOP counts for common operations
    - Add: 1 FLOP
    - Mul: 1 FLOP
    - FMA: 2 FLOP
  - FLOPs/s = Number of floating-point operations performed per second

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

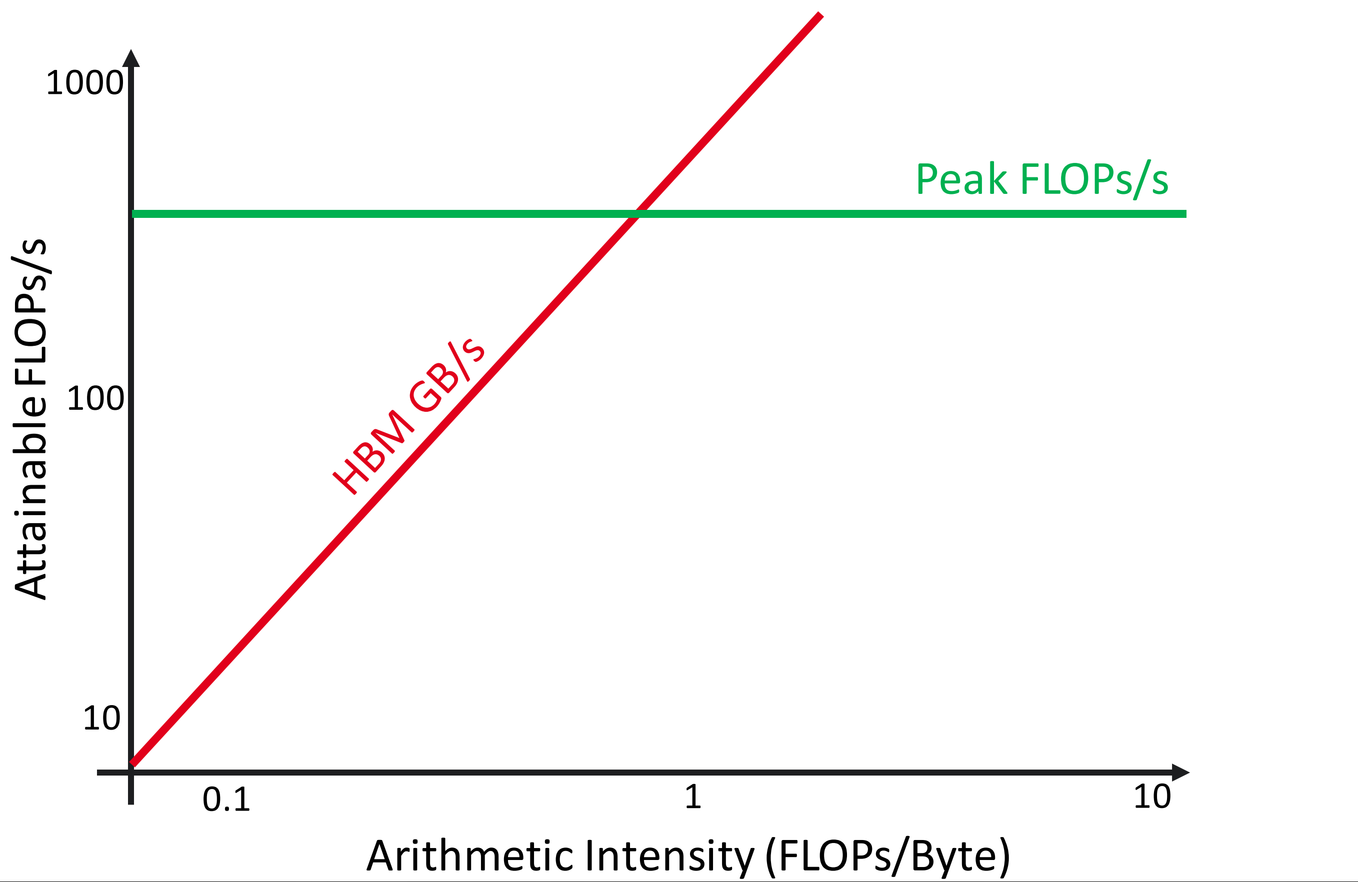
Background – What is Roofline

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



Background – What is Roofline

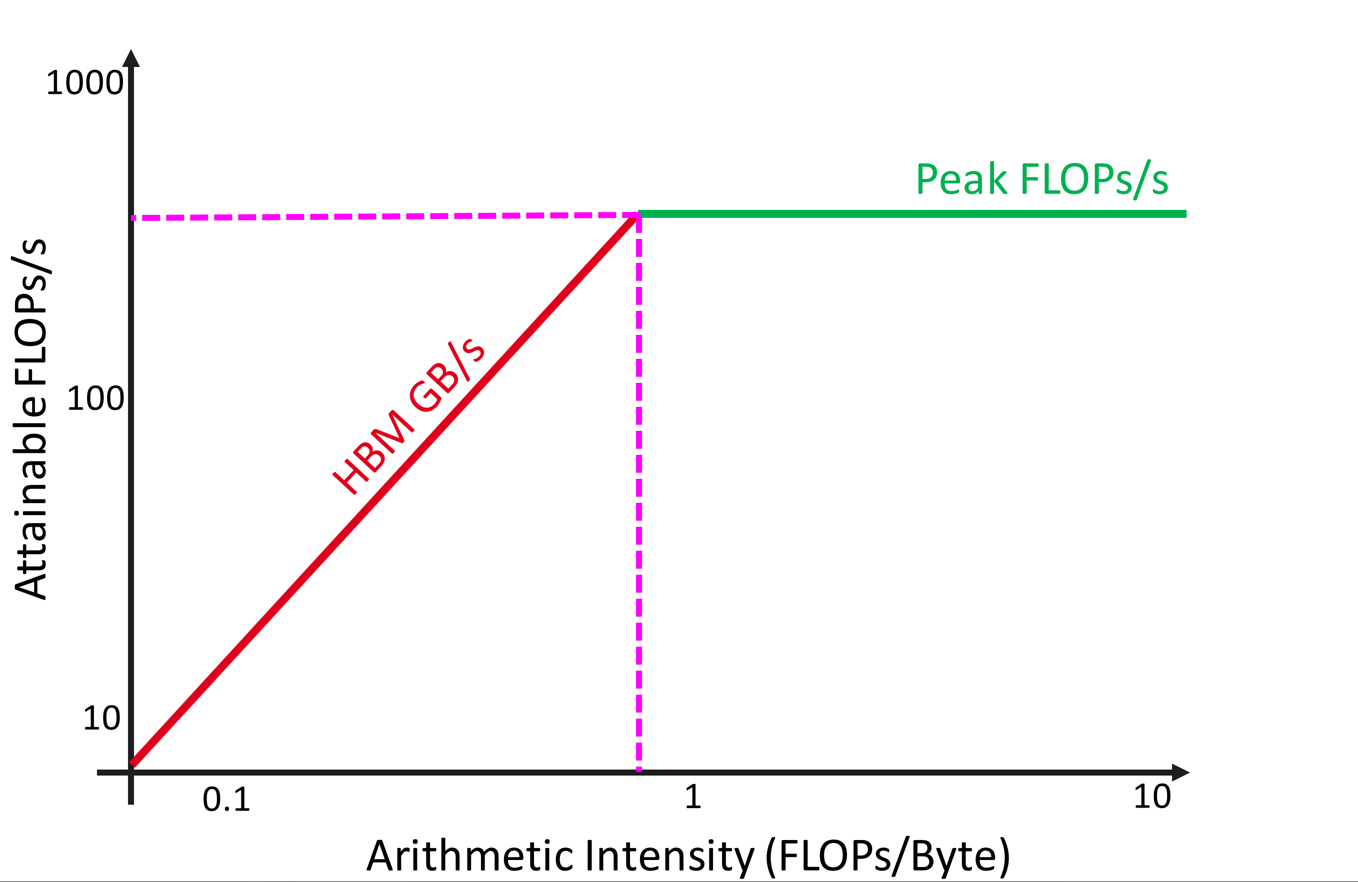
- Roofline Limiters
  - Compute
    - Peak FLOPs/s
  - Memory BW
    - AI * Peak GB/s
- Note:
  - These are empirically measured values
  - Different SKUs will have unique plots
  - Individual devices within a SKU will have slightly different plots based on thermal solution, system power, etc.
  - Omniperf uses suite of simple kernels to empirically derive these values
  - These are NOT theoretical values indicating peak performance under "unicorn" conditions



AMD

Background – What is Roofline

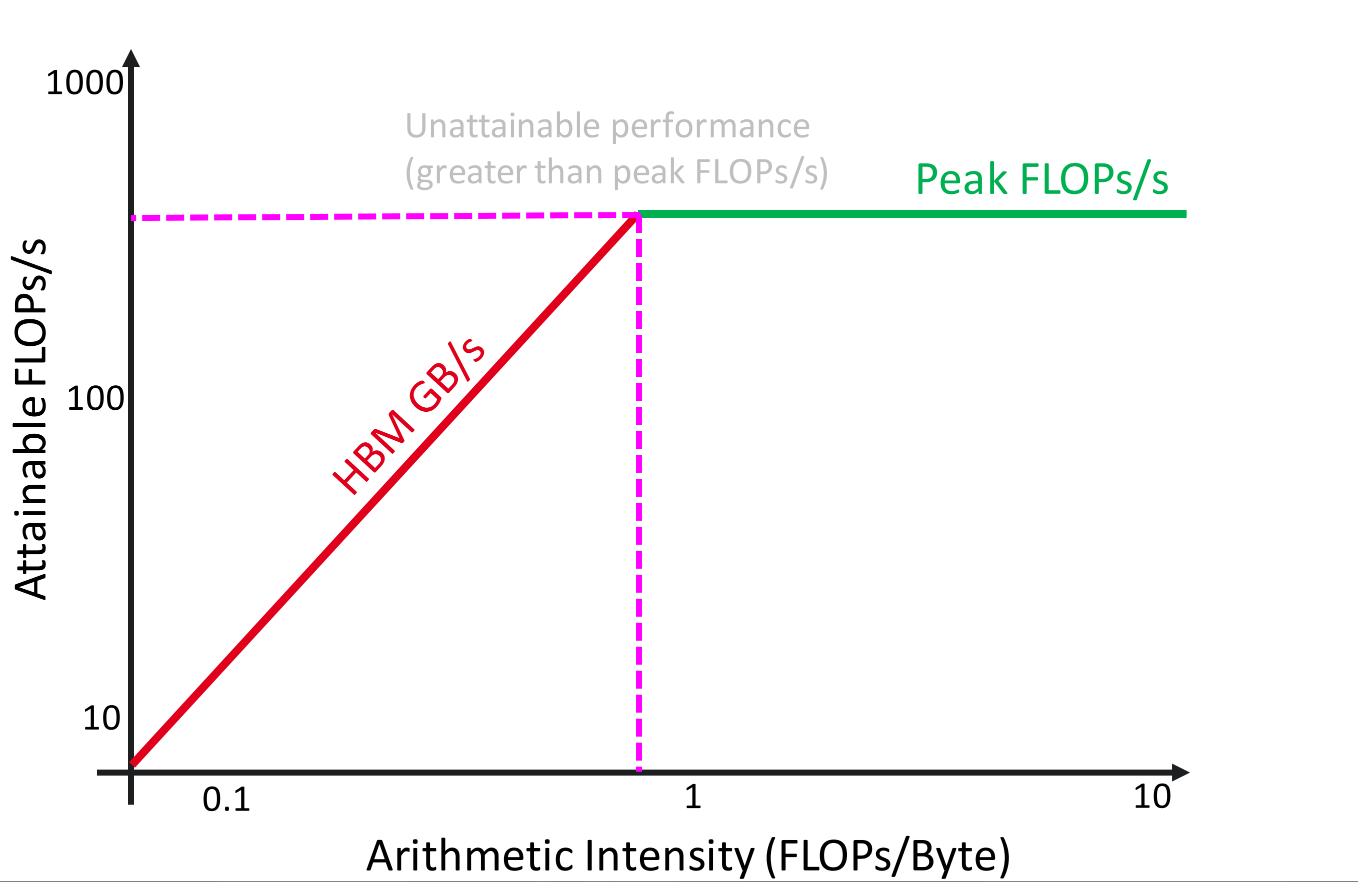
- Attainable FLOPs/s =
  - $min \begin{cases} Peak\ FLOPs/s \\ AI * Peak\ GB/s \end{cases}$
- Machine Balance:
  - Where $AI = \frac{Peak\ FLOPs/s}{Peak\ GB/s}$
  - Typical machine balance: 5-10 FLOPs/B
    - **40-80** FLOPs per double to exploit compute capability
  - MI250x machine balance: ~16 FLOPs/B
    - **128** FLOPs per double to exploit compute capability



AMD

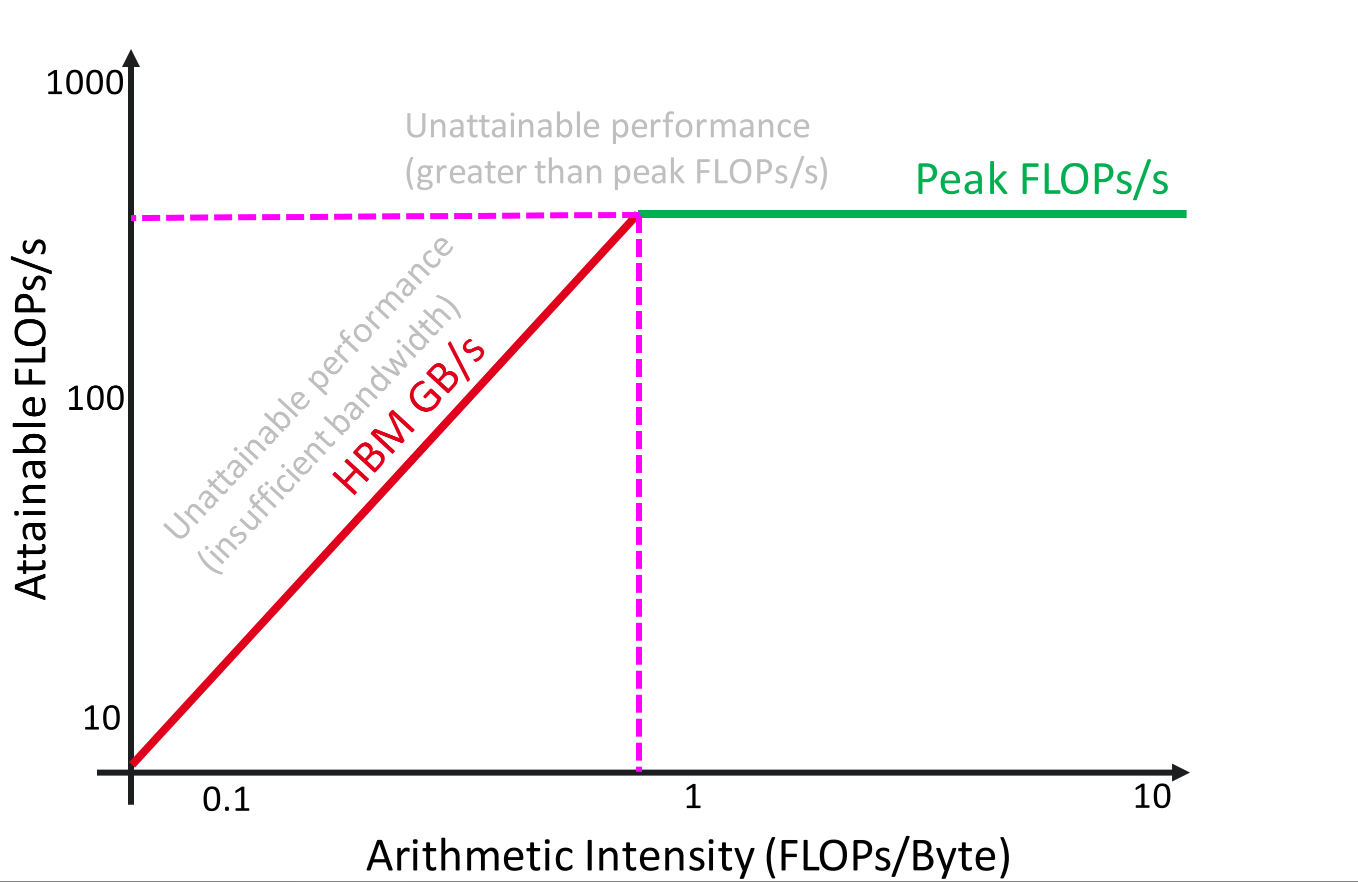
Background – What is Roofline

- 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



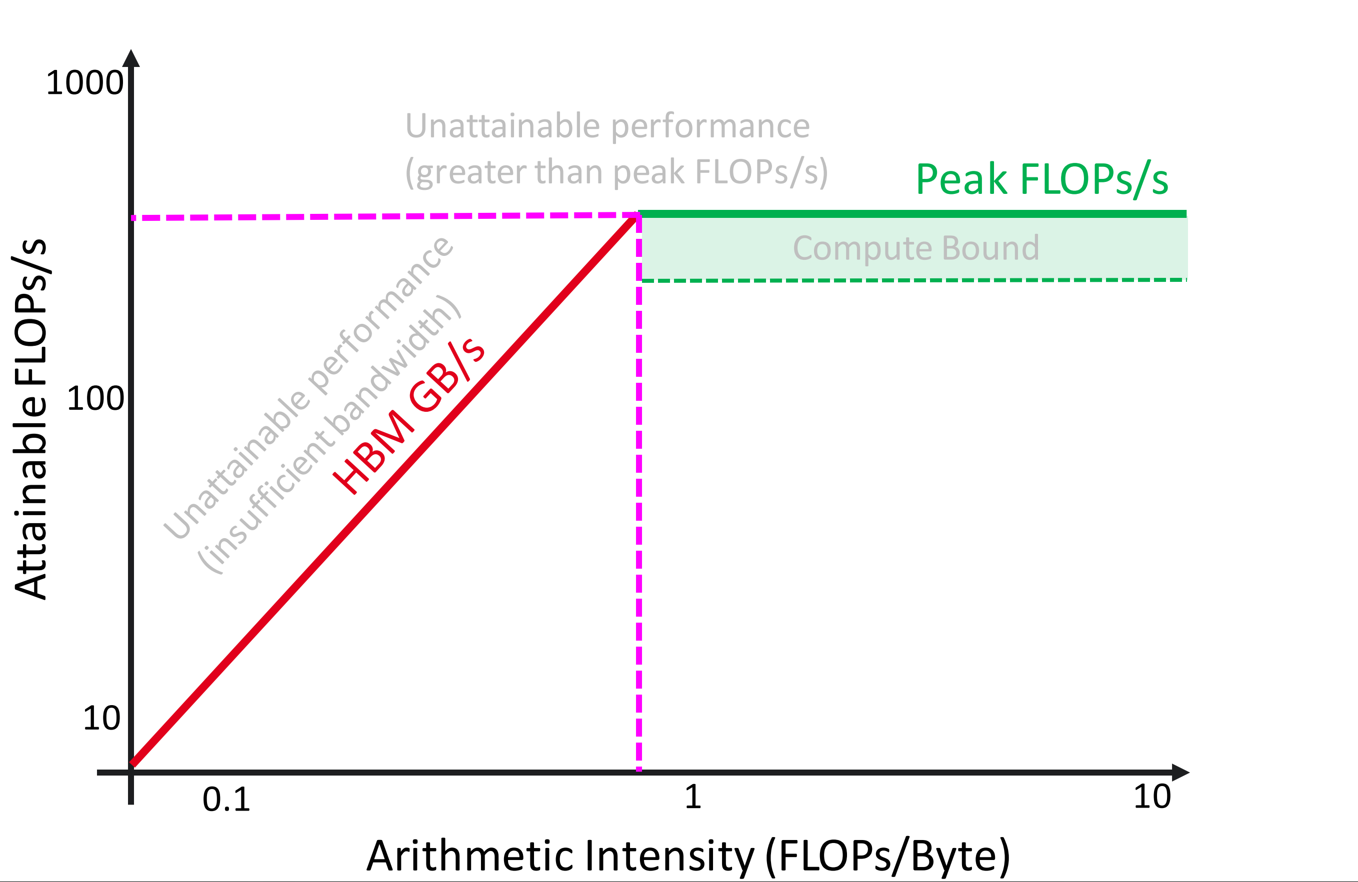
Background – What is Roofline

- 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



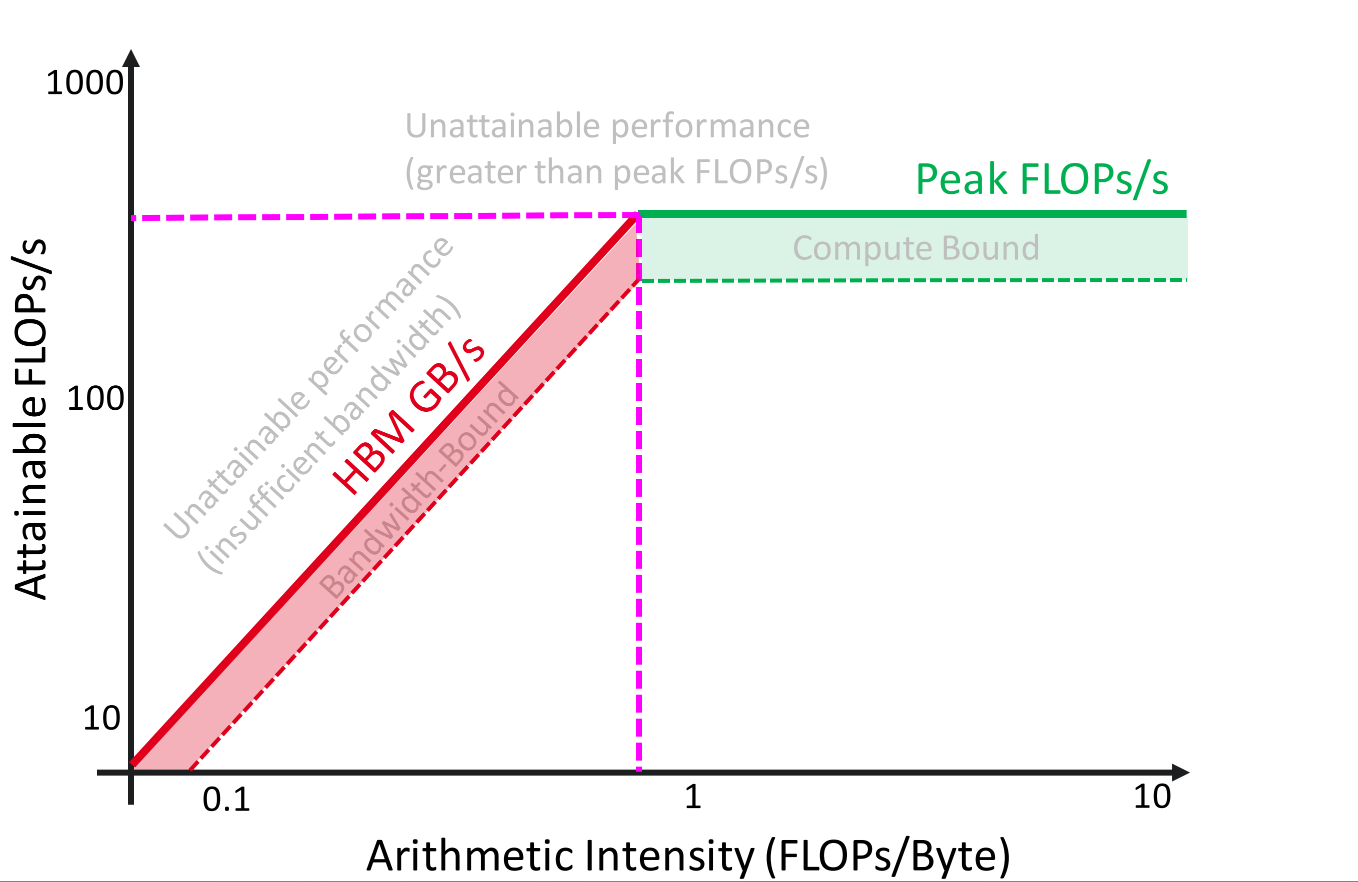
Background – What is Roofline

- 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



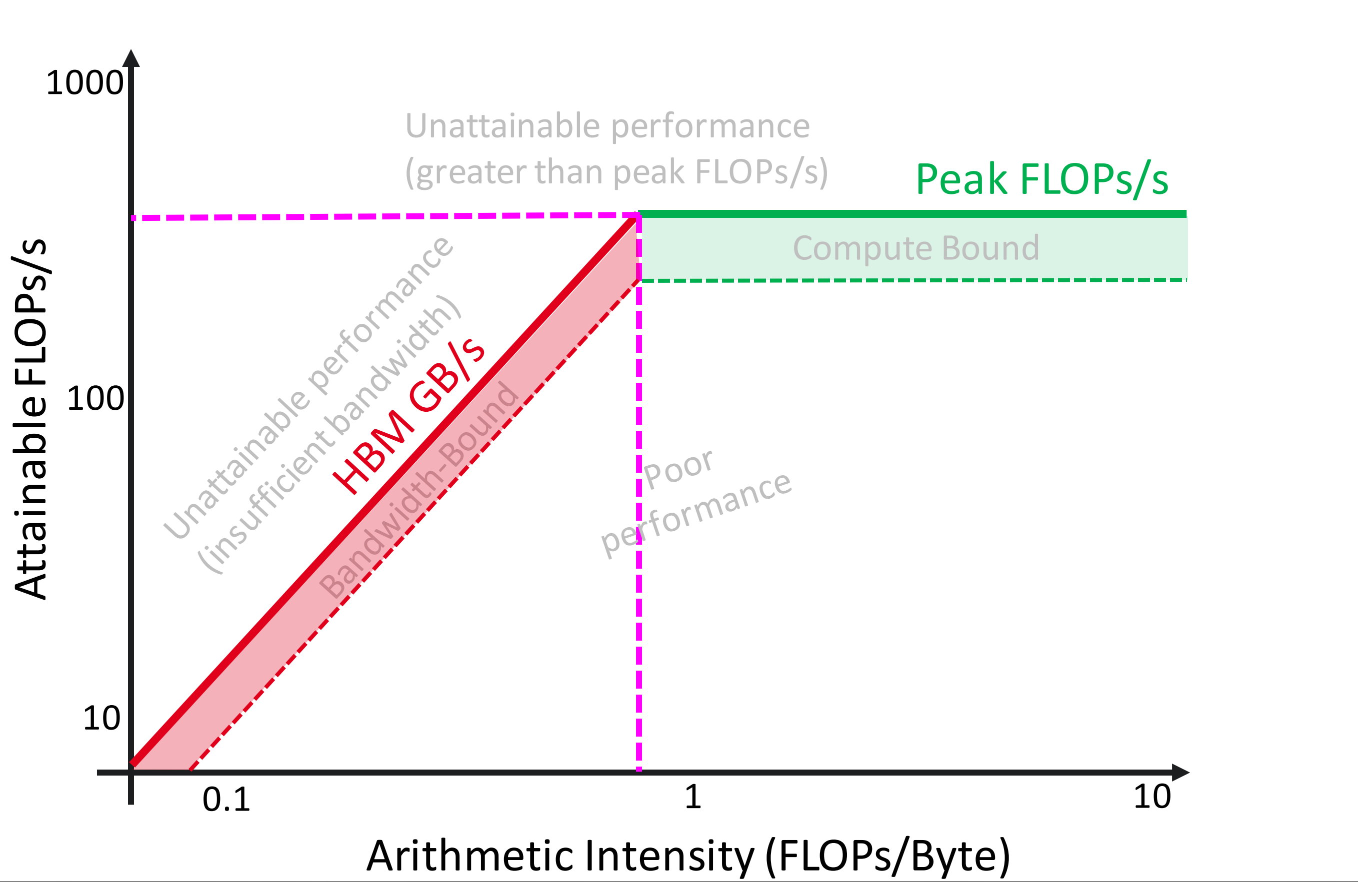
Background – What is Roofline

- 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



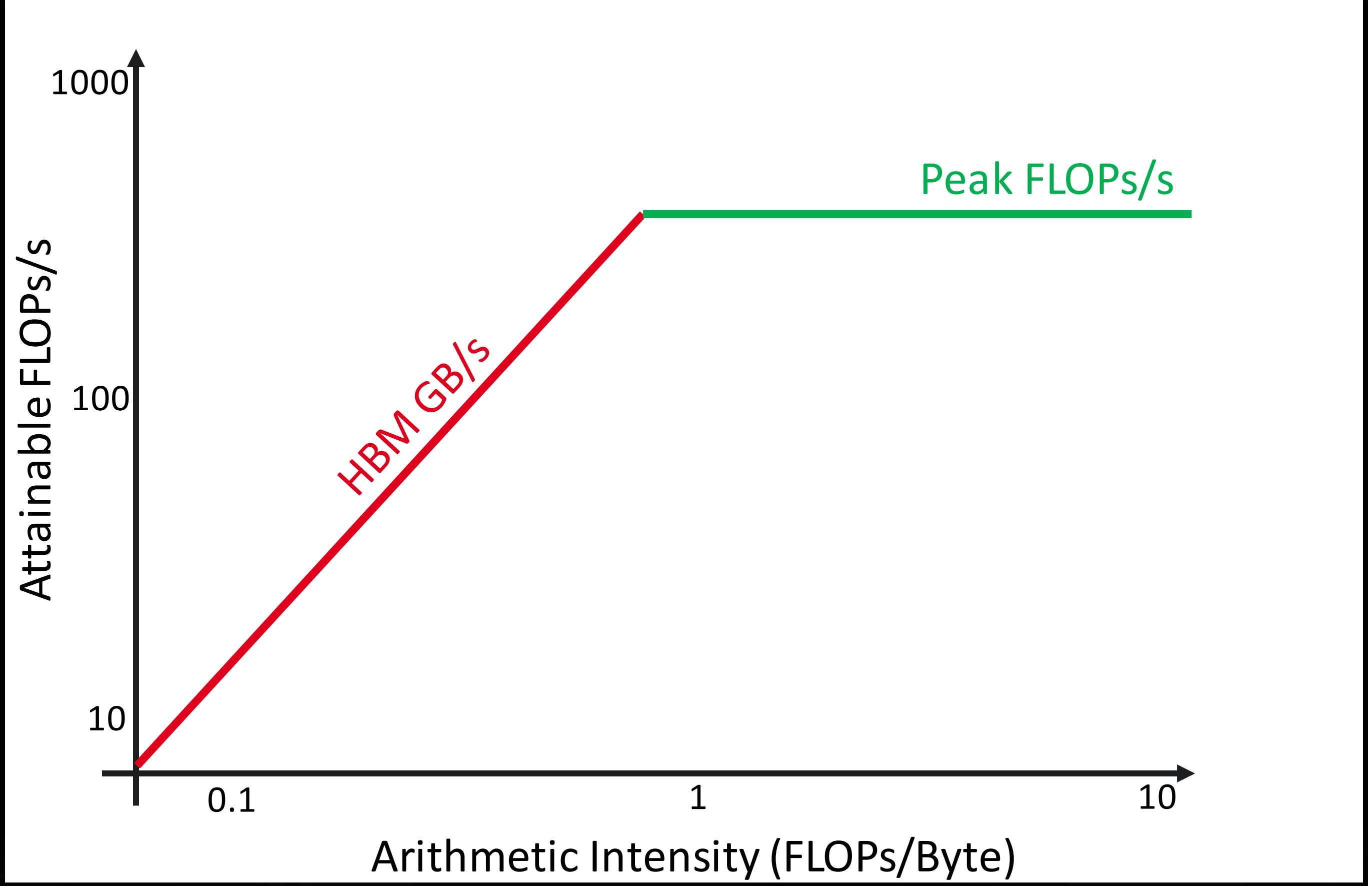
Background – What is Roofline

- 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



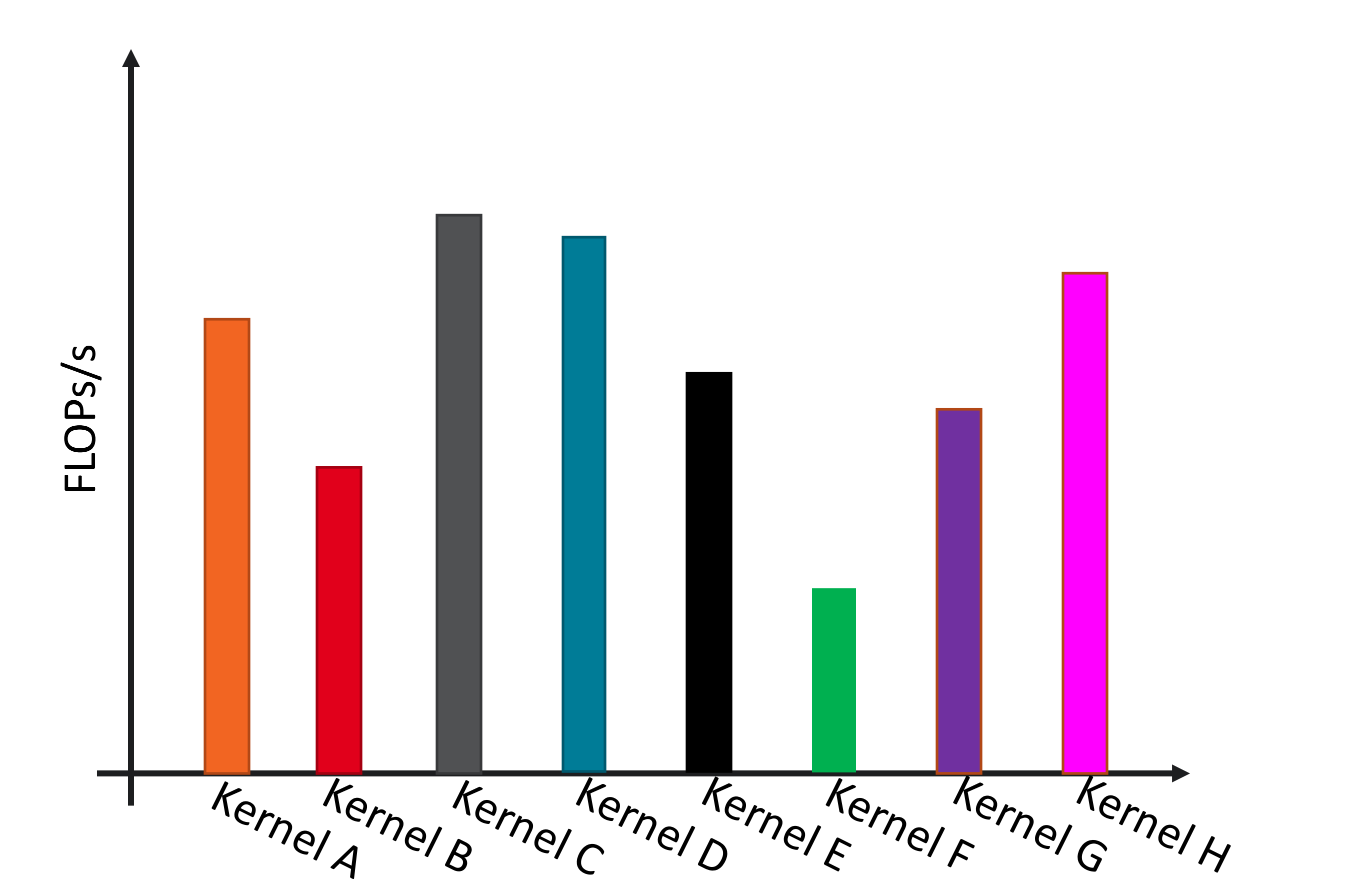
Background – What is Roofline

- Attainable FLOPs/s =
  - $min \begin{cases} Peak\ FLOPs/s \\ AI * Peak\ GB/s \end{cases}$
- 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



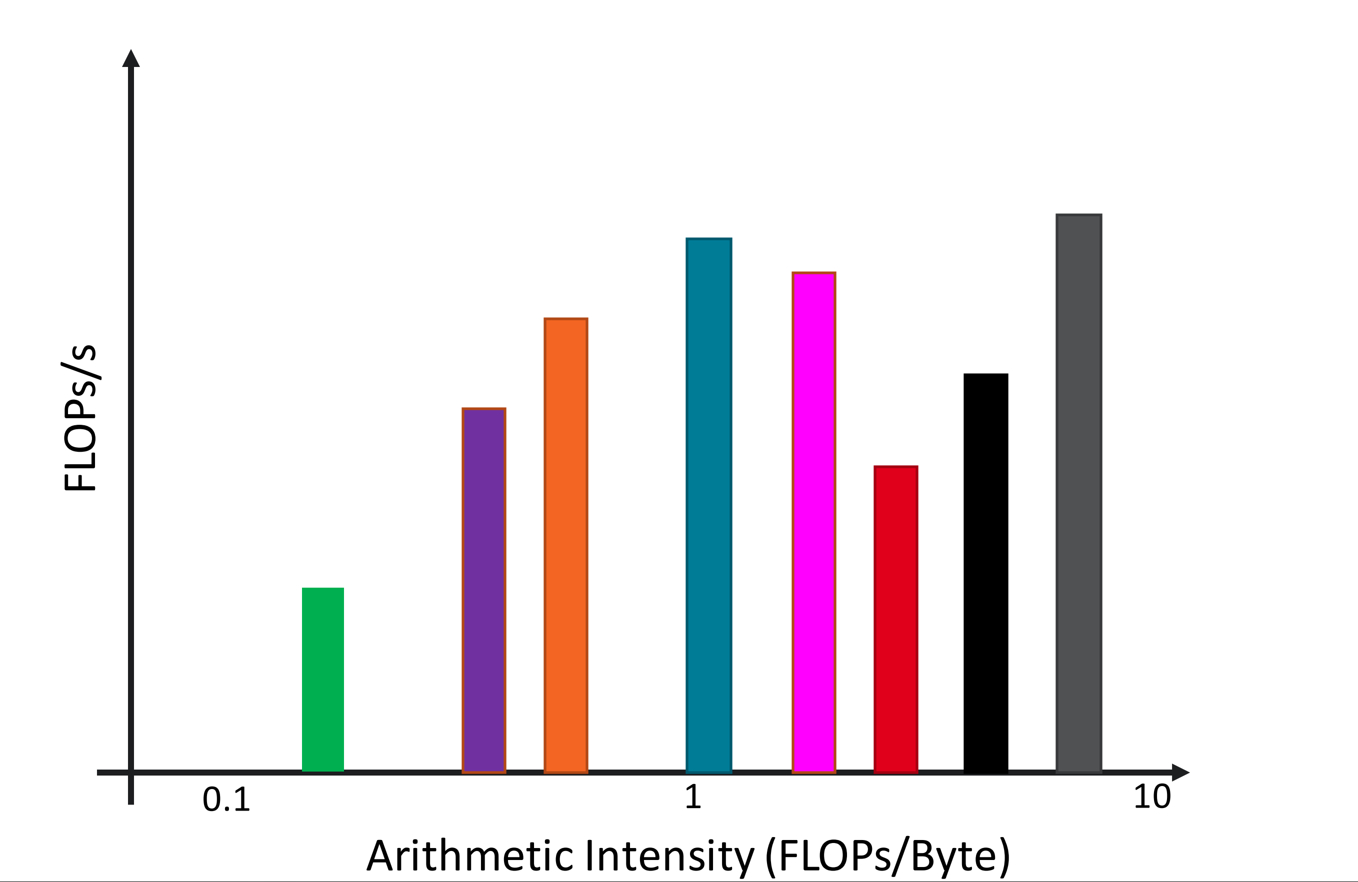
Background – What is “Good” Performance

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



Background – What is "Good" Performance

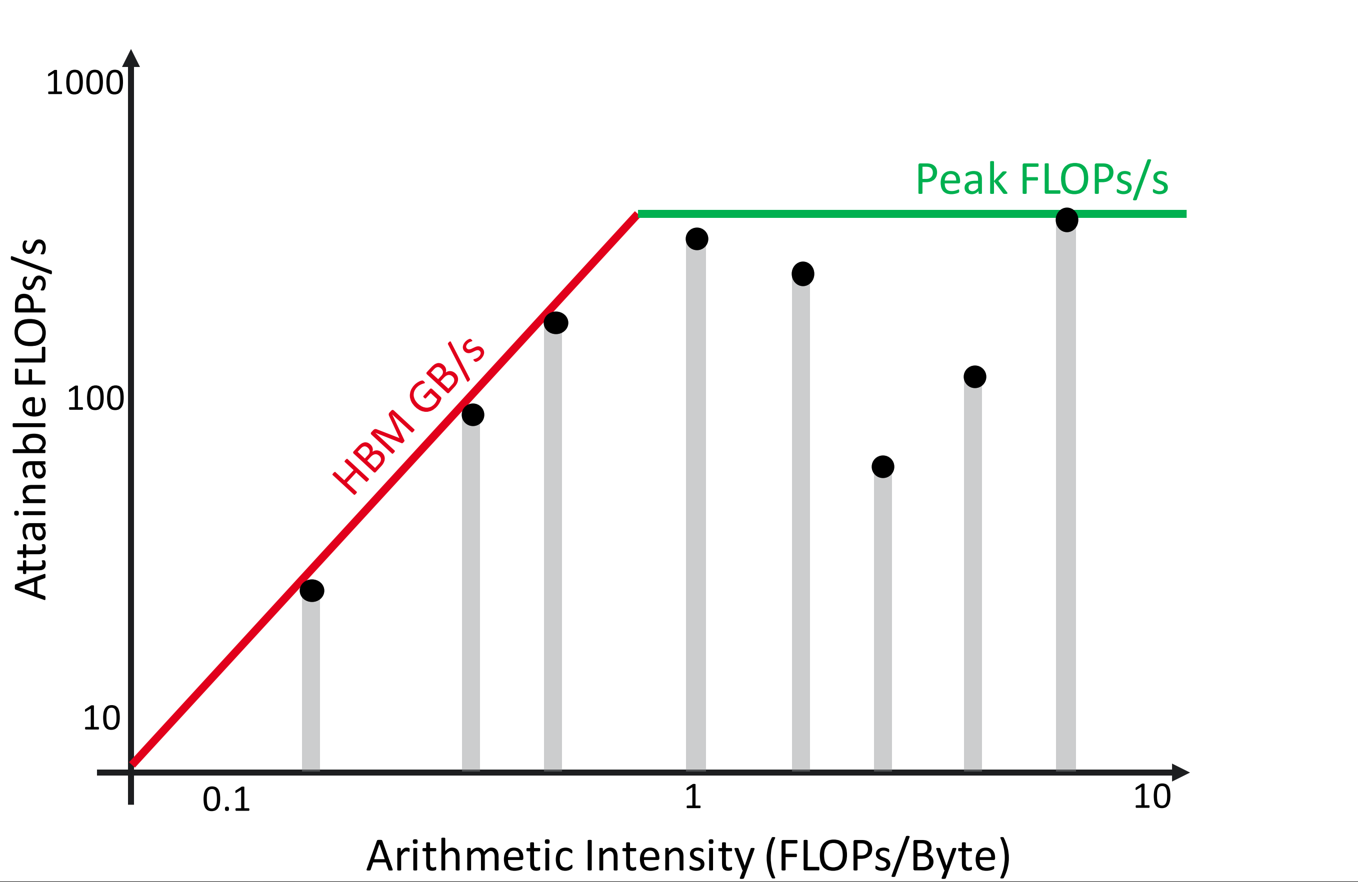
- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity



AMD

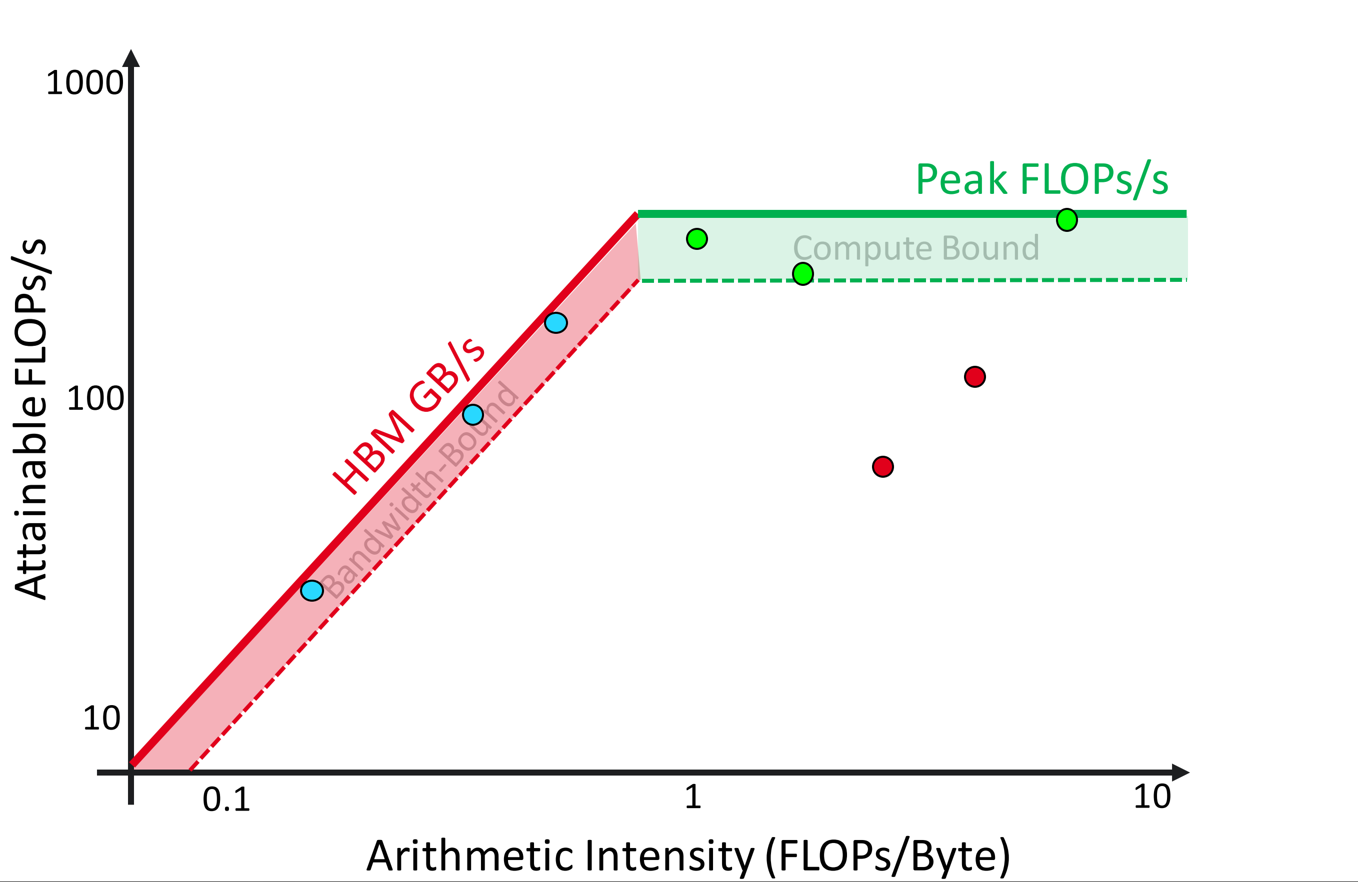
Background – What is "Good" Performance

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



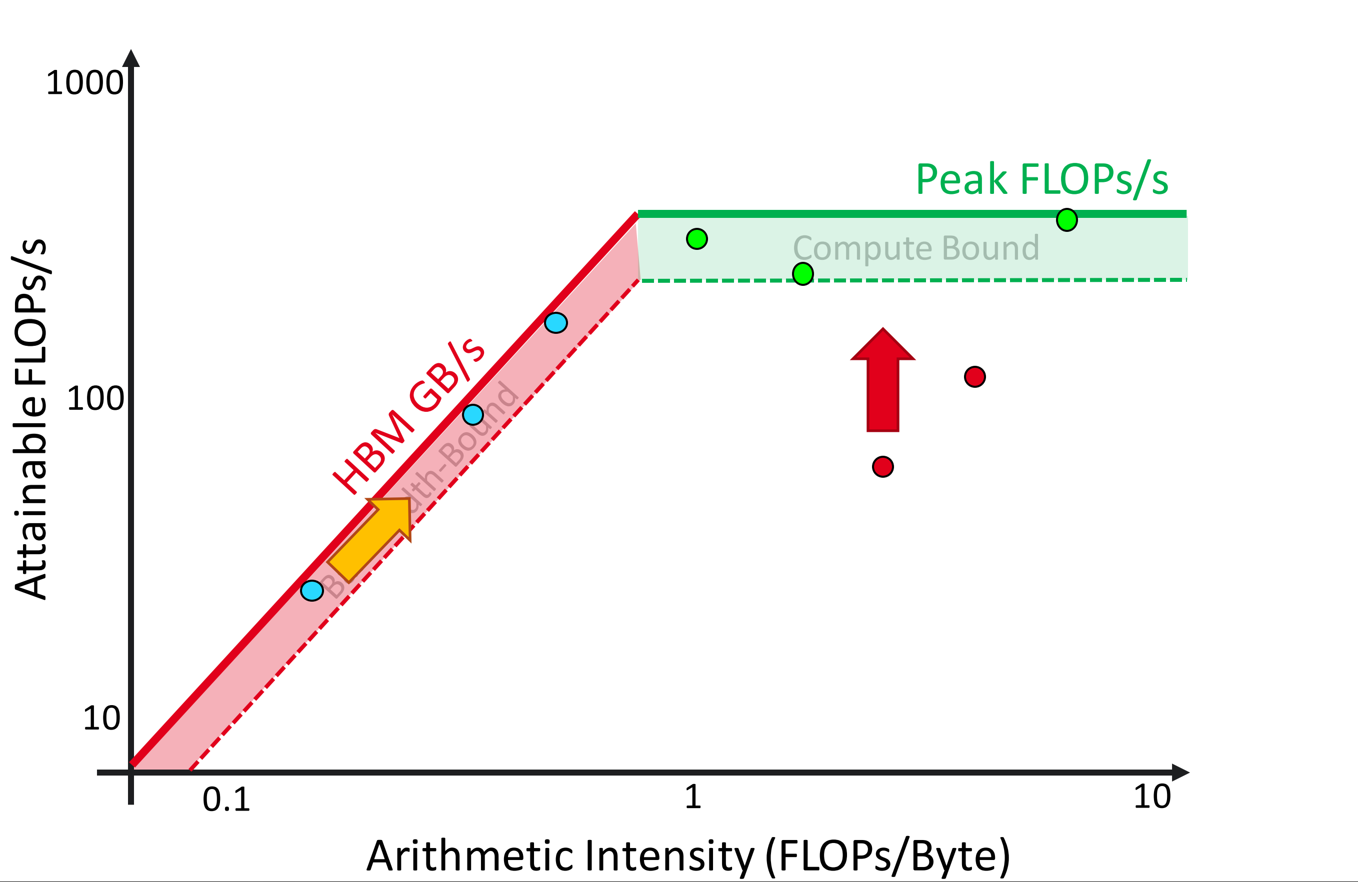
Background – What is "Good" Performance

- 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



AMD Profilers Refresher

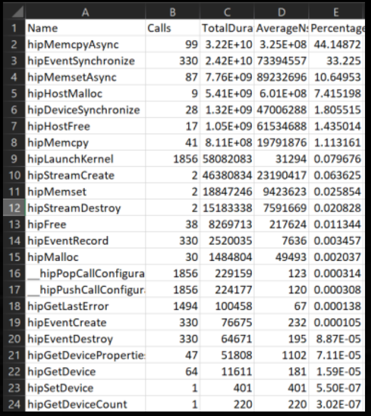
Background – AMD Profilers

- rocProfiler
  - github.com/ROCm-Developer-Tools/rocprofiler
  - Raw collection of GPU counters and traces
  - Counter collection driven by user provided input files
  - Counter results output in CSV
  - Trace collection support for:
    - HIP
    - HSA
    - GPU
  - Traces visualized with Perfetto

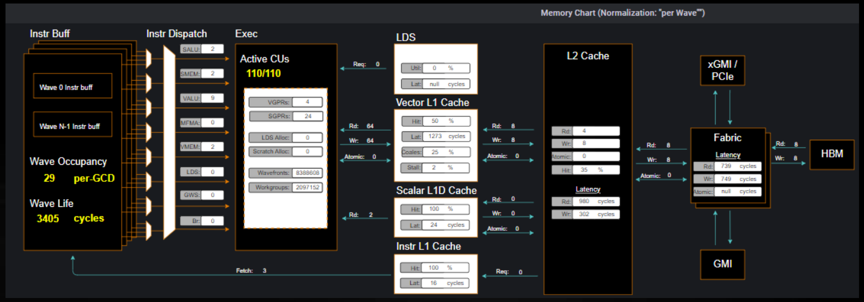
- Omnitrace
  - github.com/AMDResearch/omnitrace
  - Comprehensive trace collection and visualization of CPU+GPU
  - Includes support for:
    - HIP, HSA, GPU
    - OpenMP
    - MPI
    - Kokkos
    - Pthreads
    - Multi-GPU
  - Visualizations with Perfetto

Focus for this presentation

- **Omniperf**
  - github.com/AMDResearch/omniperf
  - Automated collection, analysis and visualization of performance counters
  - Includes support for:
    - GPU Speed-of-Light Analysis
    - Memory Chart Analysis
    - Roofline Analysis
    - Kernel comparison
  - Visualizations with Grafana

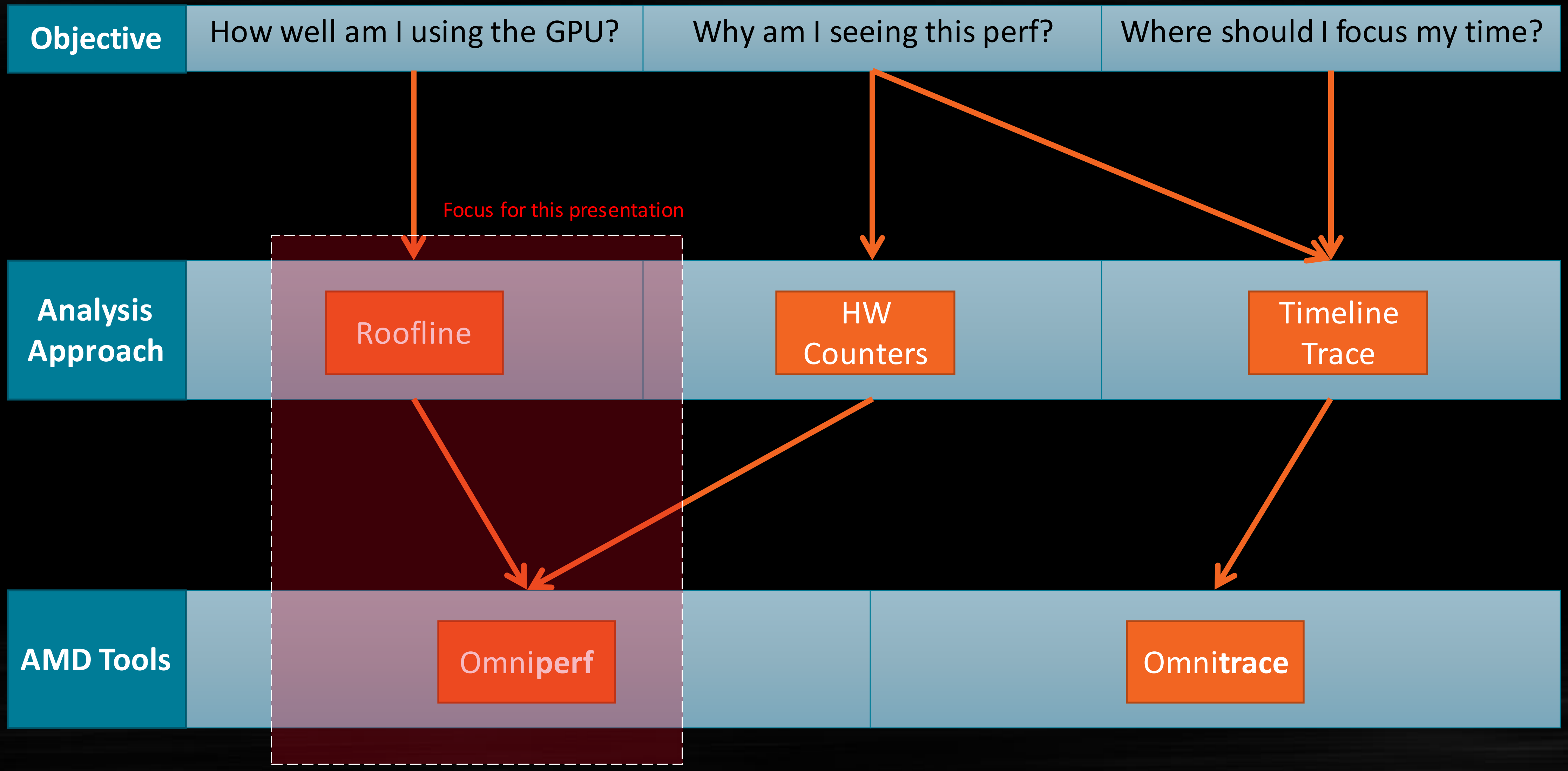


	A	B	C	D	E
1	Name	Calls	TotalDura	AverageN:	Percentage
2	hipMemcpyAsync	99	3.22E+10	3.25E+08	44.14872
3	hipEventSynchronize	330	2.42E+10	73394557	33.225
4	hipMemsetAsync	87	7.76E+09	89232696	10.64953
5	hipHostMalloc	9	5.41E+09	6.01E+08	7.415198
6	hipDeviceSynchronize	28	1.32E+09	47006288	1.805515
7	hipHostFree	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



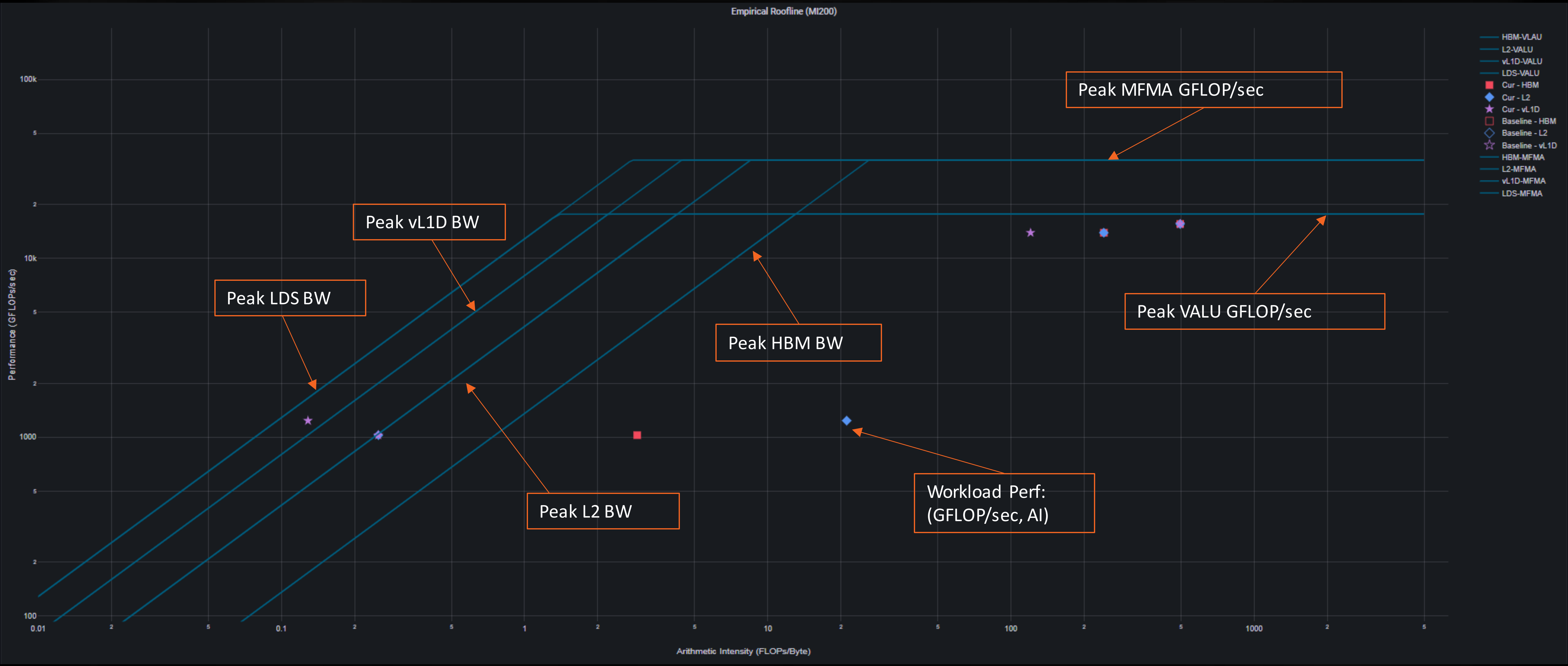
AMD

Background – AMD Profilers



Roofline Calculations on AMD Instinct™ MI200 GPUs

Empirical Hierarchical Roofline on MI200 - Overview



AMD

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 - 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

$$Total\_FLOP = 64 * (SQ\_INSTS\_VALU\_ADD\_F16 + SQ\_INSTS\_VALU\_MUL\_F16 + SQ\_INSTS\_VALU\_TRANS\_F16 + 2 * SQ\_INSTS\_VALU\_FMA\_F16)$$
$$+ 64 * (SQ\_INSTS\_VALU\_ADD\_F32 + SQ\_INSTS\_VALU\_MUL\_F32 + SQ\_INSTS\_VALU\_TRANS\_F32 + 2 * SQ\_INSTS\_VALU\_FMA\_F32)$$
$$+ 64 * (SQ\_INSTS\_VALU\_ADD\_F64 + SQ\_INSTS\_VALU\_MUL\_F64 + SQ\_INSTS\_VALU\_TRANS\_F64 + 2 * SQ\_INSTS\_VALU\_FMA\_F64)$$
$$+ 512 * SQ\_INSTS\_VALU\_MFMA\_MOPS\_F16$$
$$+ 512 * SQ\_INSTS\_VALU\_MFMA\_MOPS\_BF16$$
$$+ 512 * SQ\_INSTS\_VALU\_MFMA\_MOPS\_F32$$
$$+ 512 * SQ\_INSTS\_VALU\_MFMA\_MOPS\_F64$$

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

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

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

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

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

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

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

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

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

***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

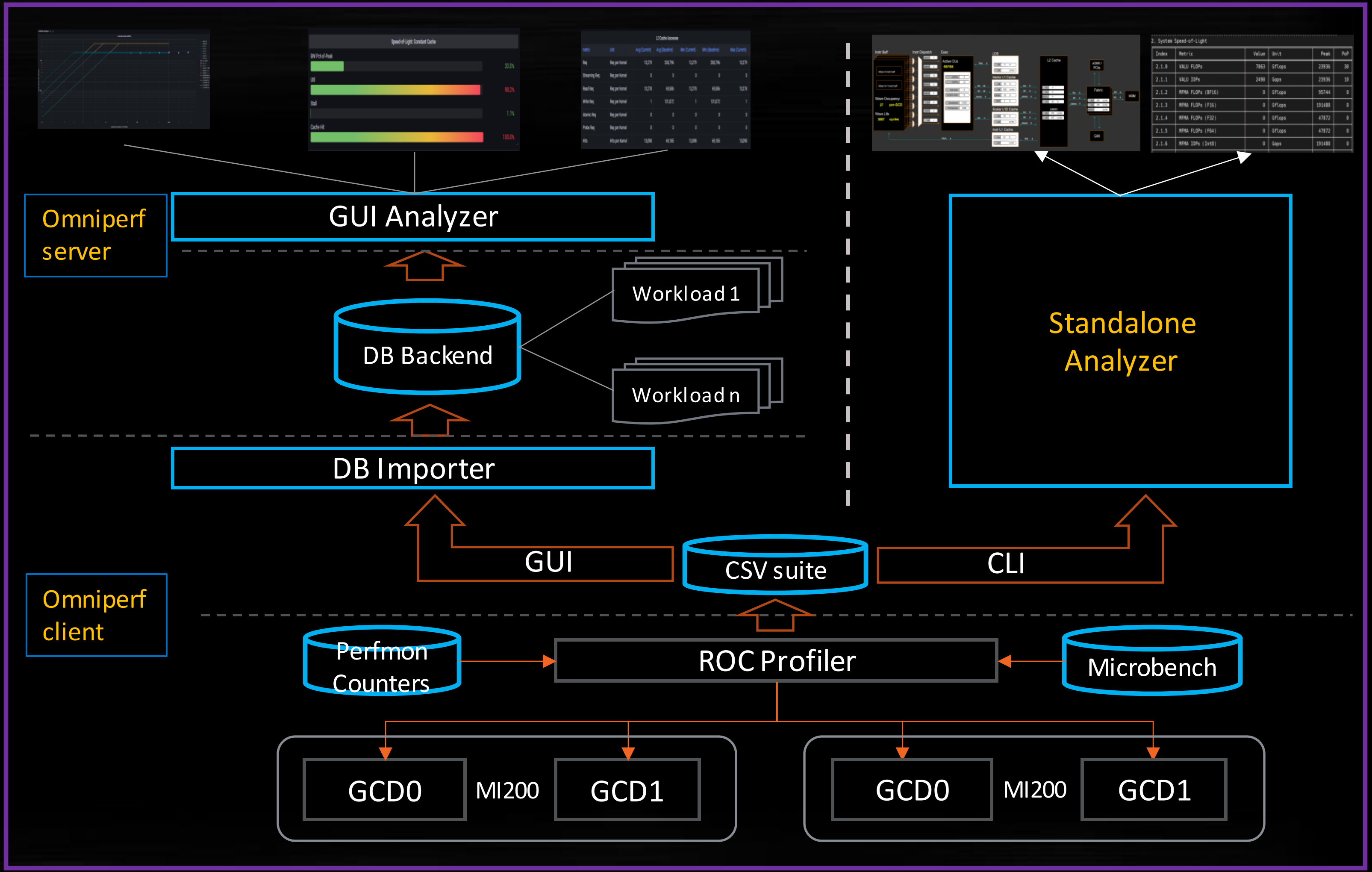
```
## roof-counters.txt

# 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
```


AMD

Omniperf Performance Analyzer

Performance Analysis on MI200 GPUs - Omniperf

- Opensource github repos
  - *https://github.com/AMDResearch/omniperf*
- Built on top of ROC Profiler
- Integrated Performance Analyzer for AMD GPUs
  - Roofline Analyzer
  - Mem Chart Analyzer
  - Speed-of-Light
  - Baseline Comparison
  - Shared Workload Database
  - Flexible Filtering and Normalization
  - Comprehensive Profiling
    - Wavefront Dispatching
    - Shader Compute
    - Local Data Share (LDS) Accesses
    - L1/L2 Cache Accesses
    - HBM Accesses
- User Interfaces
  - Grafana™ Based GUI
  - Standalone GUI



Omniperf Setup – Client

- Step 0, Setup a working ROCm™ node
  - Fresh installation: Introduction to ROCm Installation Guide for Linux (amd.com)
  - Docker® image: `sudo docker pull rocm/dev-ubuntu-20.04:5.2.3-complete`
- Step 1, Clone Omniperf repos

```
git clone https://github.com/AMDResearch/omniperf.git
```

- Step 2, Install dependencies

```
$cd omniperf
$export PATH=/global/scratch/sc2022/tools/cmake/bin:$PATH
$export INSTALL_DIR=/global/scratch/sc2022/tools/omniperf
$python3 -m pip install --system -t ${INSTALL_DIR}/python-libs -r requirements.txt
```

- Step 3, Install MongoDB® Community Version 5.0 matching the OS distro (e.g., Ubuntu® 20.04)

Install MongoDB Community Edition on Ubuntu — MongoDB Manual

```
$pip3 install --user pymongo
$wget -qO - https://www.mongodb.org/static/pgp/server-5.0.asc | sudo apt-key add -
$echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list
$sudo apt-get update
$sudo apt-get install -y mongodb-org
```

- Step 4, Build and install Omniperf client

```
$cd build
$cmake -DCMAKE_INSTALL_PREFIX=${INSTALL_DIR}/1.0.4 -DPYTHON_DEPS=${INSTALL_DIR}/python-libs
-DMOD_INSTALL_PATH=${INSTALL_DIR}/modulefiles ..
$make install
```

- Step 5, Sanity check

```
$ export PATH=$INSTALL_DIR/1.0.4/bin:$PATH
$export PYTHONPATH=$INSTALL_DIR/python-libs
$export
ROOFLINE_BIN=/global/scratch/sw/omniperf/roofl
ine.ubuntu18.04
$omniperf -version
  ----------------------------------------
  Omniperf version: 1.0.4 (release)
  Git revision:     065b4b7
  ----------------------------------------
```

Omniperf Setup – Server

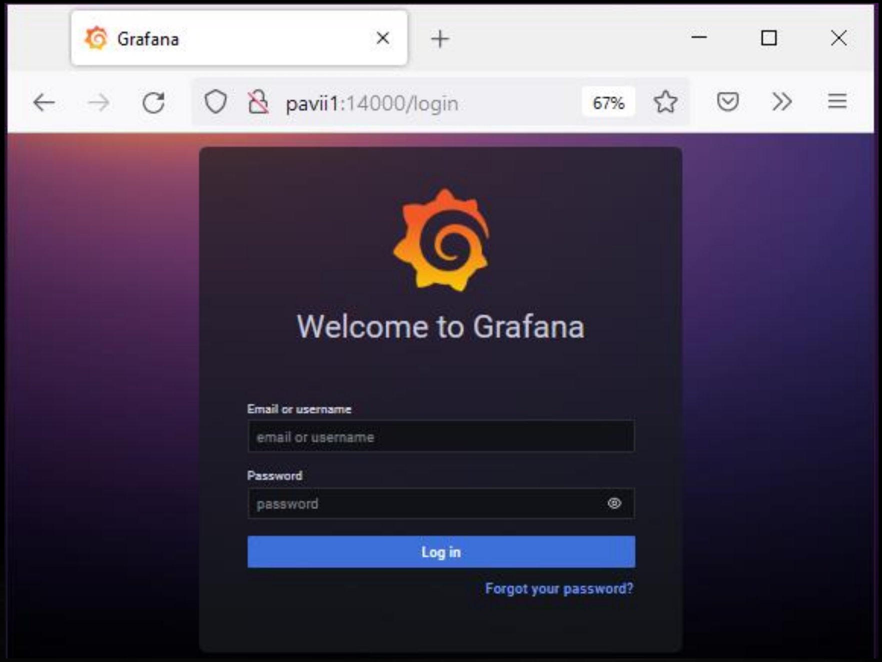
- Step 1, Setup persistent Docker® storage

```
$sudo mkdir -p /usr/local/persist
$cd /usr/local/persist/
$sudo mkdir -p grafana-storage mongodb
$sudo docker volume create --driver local --opt type=none --opt device=/usr/local/persist/grafana-storage --opt o=bind grafana-storage
$sudo docker volume create --driver local --opt type=none --opt device=/usr/local/persist/mongodb --opt o=bind grafana-mongo-db
```

- Step 2, Start the Omniperf server

```
$sudo docker-compose build
$sudo docker-compose up -d
```

- Step 3, Server Configuration
  - Refer to https://amdresearch.github.io/omniperf/



Omniperf Helloworld – vcopy Profiling

- Step 1, Workload compilation

```
$mkdir test && cd test
$cp $OMNIPERF_HOME/sample/vcopy.cpp .
$hipcc vcopy.cpp -o vcopy
$./vcopy 1048576 256
  Finished allocating vectors on the CPU
  Finished allocating vectors on the GPU
  Finished copying vectors to the GPU
  sw thinks it moved 1.000000 KB per wave
  Total threads: 1048576, Grid Size: 4096 block Size:256, Wavefronts:16384:
  Launching the  kernel on the GPU
  Finished executing kernel
  Finished copying the output vector from the GPU to the CPU
  Releasing GPU memory
  Releasing CPU memory
```

- Step 2, Workload profiling

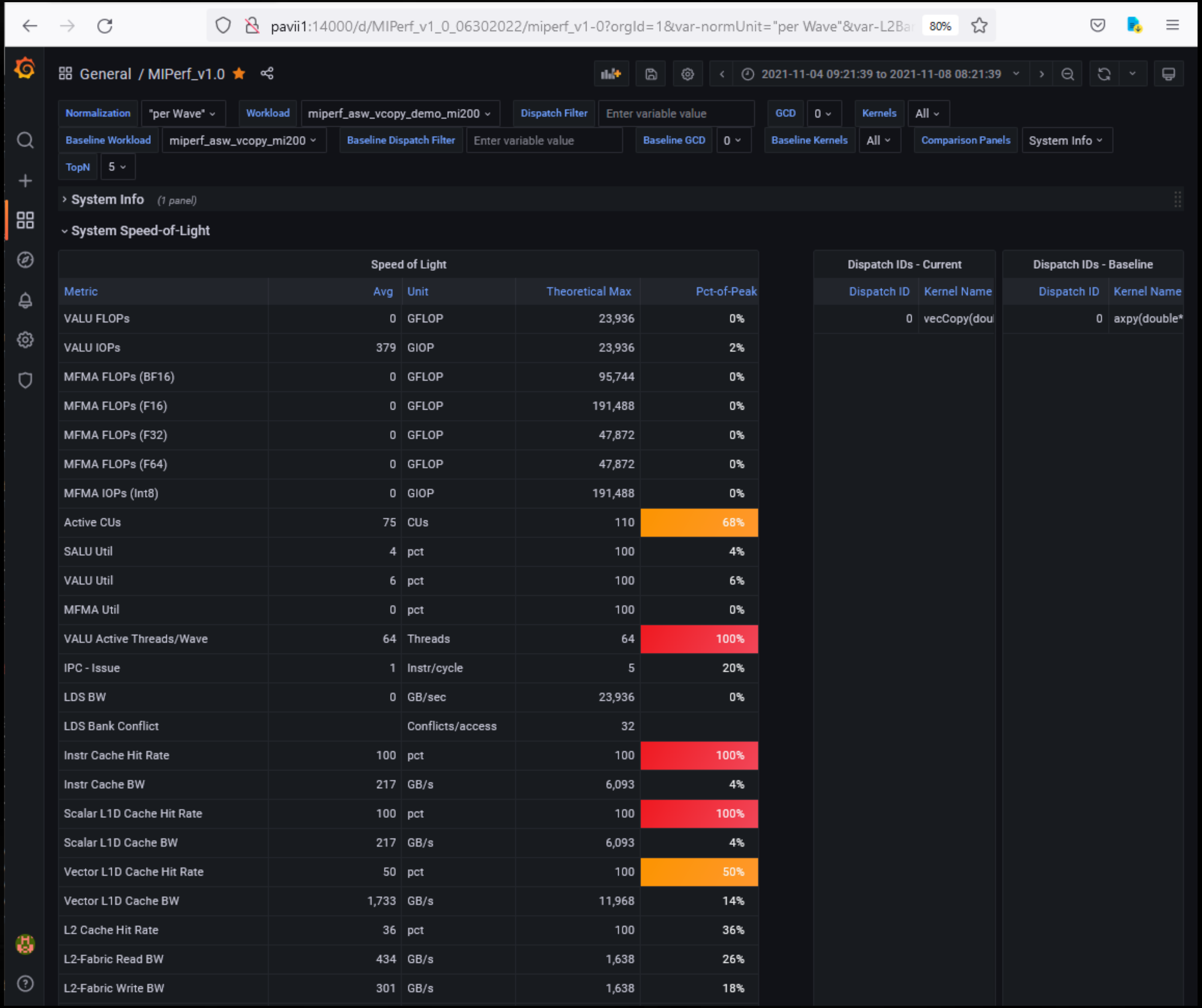
```
$omniperf profile -n vcopy_demo -- ./vcopy 1048576 256
```

Omniperf Helloworld – vcopy Profiling (Cont'd)

- Step 3, Import profiling results

```
$omniperf database --import -H pavii1 -u amd -t asw -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! --
```

- Step 4, Analyze workload performance



Speed of Light				
Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
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%

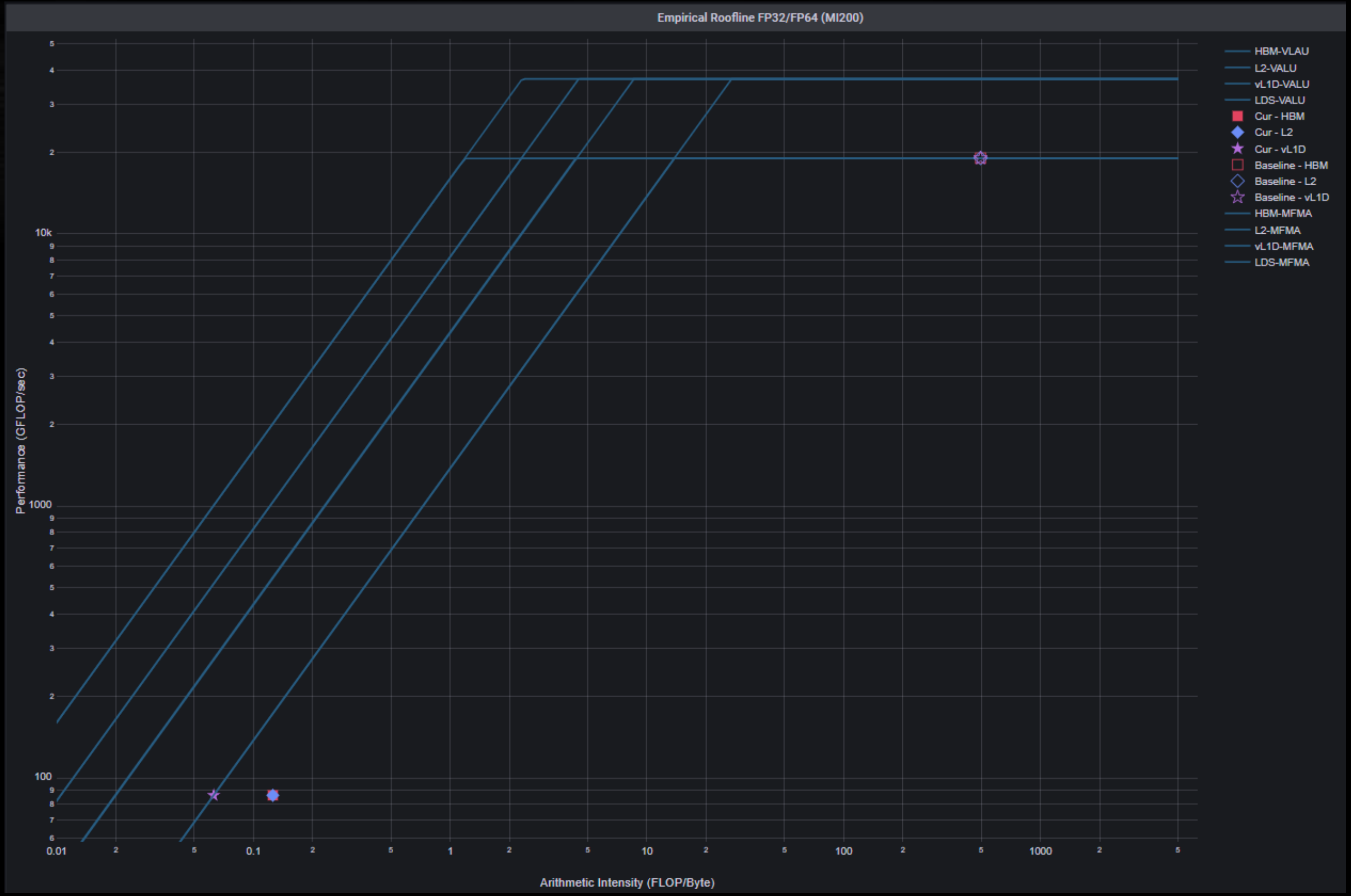
Dispatch IDs - Current	
Dispatch ID	Kernel Name
0	vecCopy(dou

Dispatch IDs - Baseline	
Dispatch ID	Kernel Name
0	axpy(double*

[Public]

Roofline Based Performance Analysis

- 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

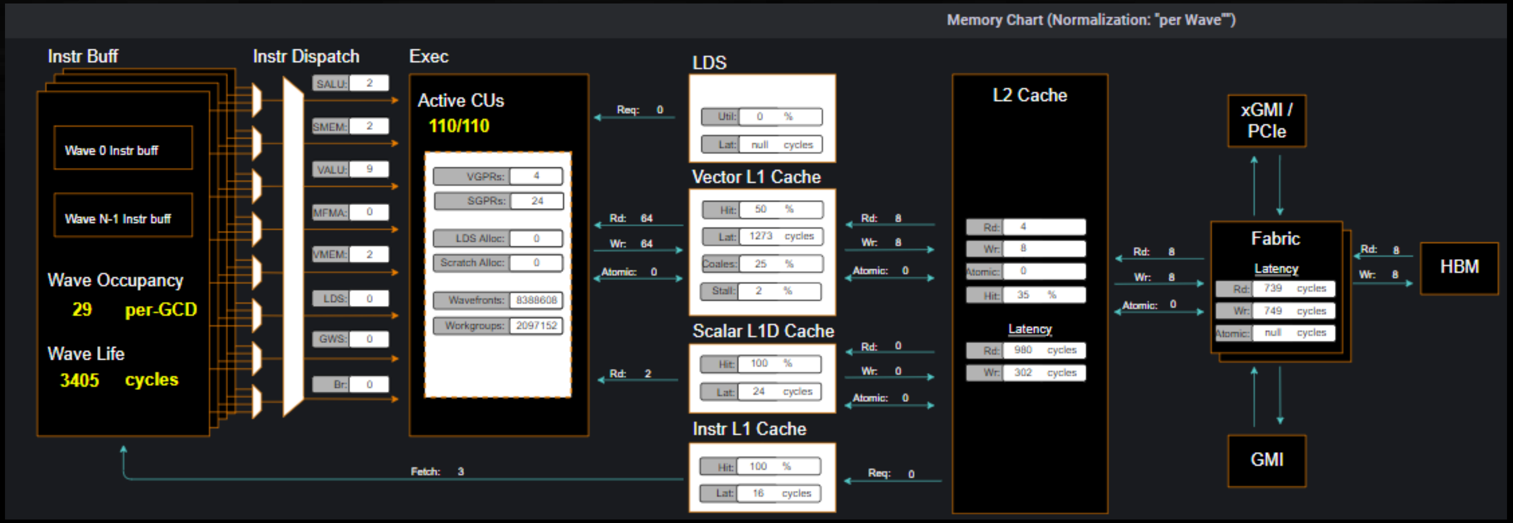


Top Kernels

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

Roofline Based Performance Analysis (Cont'd)

- SoC Performance Overview – initial assessment
  - Instruction/data flow
  - Speed-of-light
- Omniperf tooling support
  - System Speed-of-Light
  - Mem-chart view
  - Kernel statistics



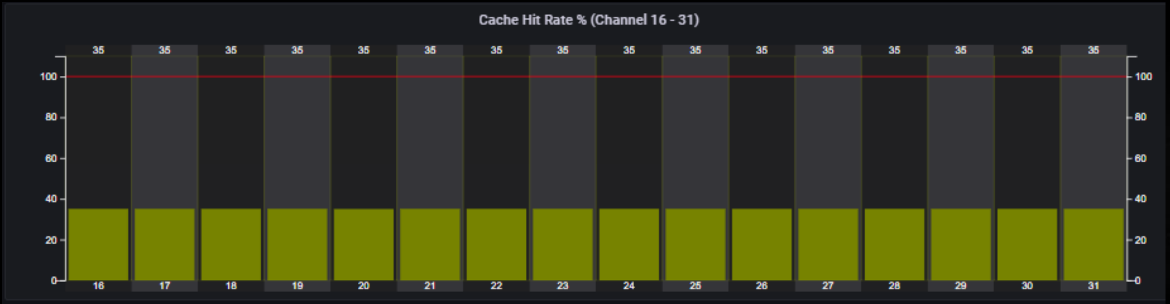
Speed of Light				
Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
VALU FLOPs	0	GFLOP	23,936	0%
VALU IOPs	433	GIOP	23,936	2%
MFMA FLOPs (BF16)	0	GFLOP	95,744	0%
MFMA FLOPs (F16)	0	GFLOP	191,488	0%
MFMA FLOPs (F32)	0	GFLOP	47,872	0%
MFMA FLOPs (F64)	0	GFLOP	47,872	0%
MFMA IOPs (Int8)	0	GIOP	191,488	0%
Active 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%

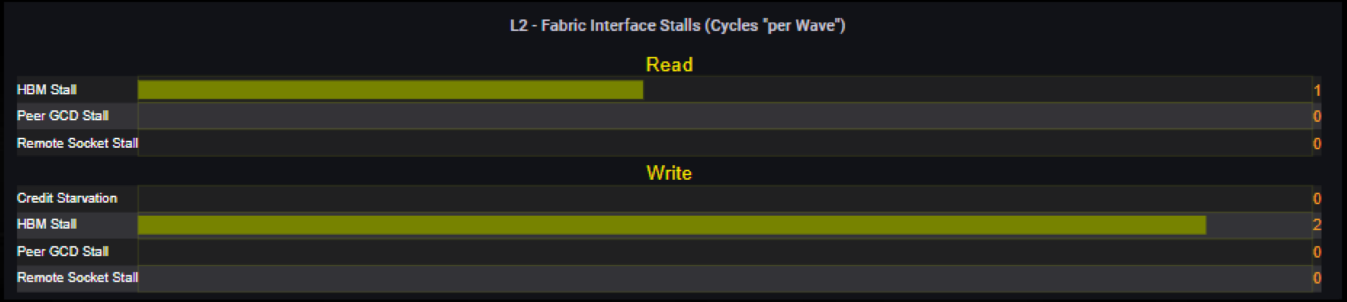
Roofline Based Performance Analysis (Cont'd)

- L2 Cache and HBM Data Accesses
  - Transactions
  - Bandwidth
  - Latency
  - Stalls
- Omniperf tooling support
  - L2 Cache speed-of-light
  - L2-Fabric metrics
    - Bandwidth
    - Latency
    - Stall
  - Per-channel L2 metrics

L2 - Fabric Transactions

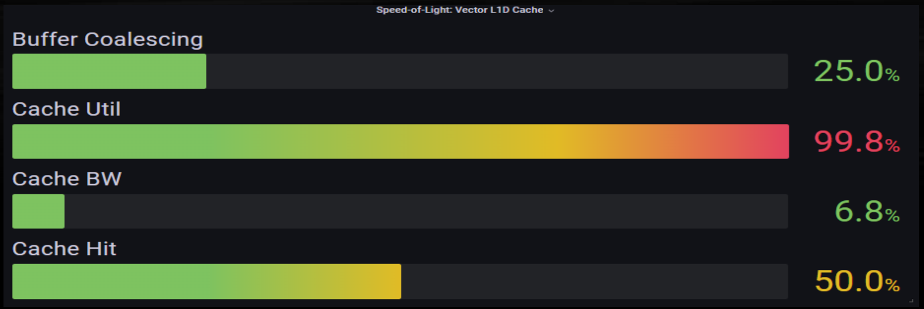
Metric	Avg	Min	Max	Unit
Read BW	693,148,700,953	664,565,016,054	695,197,543,698	Bytes per Sec
Write BW	692,659,558,092	664,096,634,666	694,705,946,653	Bytes per Sec
Read (32B)	0	0	0	Req per Sec
Read (Uncached 32...	2,304,240	1,434,649	2,370,898	Req per Sec
Read (64B)	10,830,448,452	10,383,828,376	10,862,461,620	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 32...	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





Roofline Based Performance Analysis (Cont'd)

- Vector L1D Cache Accesses
  - cache hit/miss
  - Cache BW
  - Buffer coalescing
  - L1D – L2 cache transactions
- Omniperf tooling support
  - Vector L1D Cache speed-of-light
  - Vector L1D Cache access metrics
  - Vector L1D Cache stalls
    - Tag RAM access stalls
    - Vector L1D stalled waiting for L2 data



Vector L1D Cache Accesses

metric	avg	min	max	Unit
Total Req	128	128	128	Req per Wave
Read Req	64	64	64	Req per Wave
Write Req	64	64	64	Req per Wave
Atomic Req	0	0	0	Req per Wave
Cache Accesses	32	32	32	Req per Wave
Cache Hits	16	16	16	Req per Wave
Cache Hit Rate	50	50	50	pct
Invalidate	0	0	0	per Wave
L1-TCR Read	8	8	8	Req per Wave
L1-L2 Write	8	8	8	Req per Wave
L1-L2 Atomic	0	0	0	Req per Wave
L1 Access Latency	1,273	1,240	1,341	Cycles
L1-L2 Read Latency	980	961	1,002	Cycles
L1-L2 Write Latency	302	294	312	Cycles

Vector L1D Cache Stalls

Metric	Mean	Min	Max	unit
Stalled on L2 Data	80.2%	79.0%	81.1%	pct
Stalled on L2 Req	2.5%	2.1%	2.9%	pct
Tag RAM Stall (Read)	0%	0%	0%	pct
Tag RAM Stall (Write)	0%	0%	0%	pct
Tag RAM Stall (Atomic)	0%	0%	0%	pct

Roofline Based Performance Analysis (Cont'd)

- Local Data Share (LDS) Data Accesses
  - Transactions
  - Bandwidth
  - LDS bank conflict
  - Latency
- Omniperf tooling support
  - LDS speed-of-light
  - LDS metrics
    - Instructions
    - Bandwidth
    - Latency
    - Bank conflicts

LDS Stats

metric	avg	min	max	Unit
Wave Cycles	119,361	2,478	120,999	Cycles/Wave
LDS Instrs	1,981	11	2,001	Instr per Wave
Bandwidth	507,980	3,584	513,024	Bytes per Wave
Bank Conficts/Access	0	0	0	Conflicts/Access
Index Accesses	3,969	28	4,008	Cycles per Wave
Atomic Cycles	0	0	0	Cycles per Wave
Bank Conflict	0	0	0	Cycles per Wave
Addr Conflict	0	0	0	Cycles per Wave
Unaligned Stall	0	0	0	Cycles per Wave
Mem Violations	0	0	0	per Wave
LDS Latency	41	41	43	Cycles

Roofline Based Performance Analysis (Cont'd)

- Shader Compute
  - Wavefront life distribution
  - Instruction mix
  - Floating/Integer operations
  - Compute pipeline performance
  - Memory access latencies

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

Roofline Based Performance Analysis (Cont'd)

- Shader Compute
  - Wavefront life distribution
  - Instruction mix
  - Floating/Integer operations
  - Compute pipeline performance
  - Memory access latencies

MFMA Arithmetic Instr Mix

MFMA Instr	Count
MFMA-I8	0
MFMA-F16	0
MFMA-BF16	0
MFMA-F32	0
MFMA-F64	995

Roofline Based Performance Analysis (Cont'd)

- Shader Compute
  - Wavefront life distribution
  - Instruction mix
  - Floating/Integer operations
  - Compute pipeline performance
  - Memory access latencies

Arithmetic Operations

Metric	Avg	Min	Max	Unit
FLOPs (Total)	2,037,843	0	4,096,064	OPs per Wave
INT8 OPs	0	0	0	OPs per Wave
F16 OPs	0	0	0	OPs per Wave
BF16 OPs	0	0	0	OPs per Wave
F32 OPs	0	0	0	OPs per Wave
F64 OPs	2,037,843	0	4,096,064	OPs per Wave

AMD

Roofline Based Performance Analysis (Cont'd)

- Shader Compute
  - Wavefront life distribution
  - Instruction mix
  - Floating/Integer operations
  - Compute pipeline performance
  - Memory access latencies

Pipeline Stats

Metric	Avg	Min	Max	Unit
IPC (Avg)	0.388	0.151	0.625	Instr/cycle
IPC (Issue)	1	1	1	Instr/cycle
SALU Util	14.0	3.34	24.8	pct
VALU Util	10.1	7.51	12.5	pct
VALU Active Threads	64	64	64	Threads
MFMA Util	49.2	0	98.8	pct
MFMA Instr Cycles	32	32	32	cycles/instr

Memory Latencies

Metric	Avg (Current)	Min (Current)	Max (Current)	Unit
VMEM Latency	937	286	1597	Cycles
SMEM Latency	206	66	440	Cycles
Instr Fetch Latency	16	16	16	Cycles
LDS Latency				Cycles

Roofline Based Performance Analysis (Cont'd)

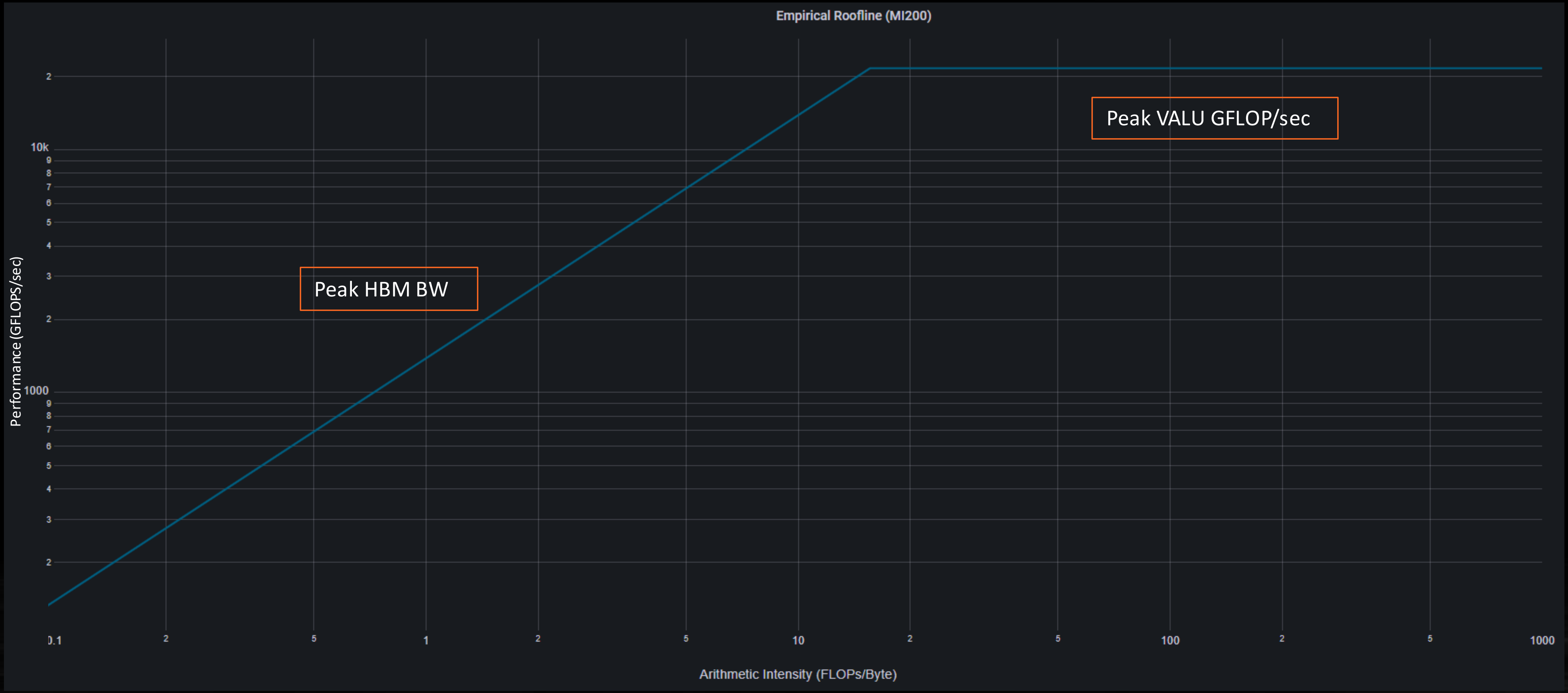
- Dispatch Bound
  - Wavefront dispatching failure due to resources limitation
    - Wavefront slots
    - VGPR
    - SGPR
    - LDS allocation
    - Barriers
    - Etc.
  - Omniperf tooling support
    - Shader Processor Input (SPI) metrics

SPI Resource Allocation				
Metric	Avg	Min	Max	Unit
Wave request Failed (CS)	2,419,396	113,061	3,399,120	Cycles
CS Stall	1,003,075	287,240	2,007,747	Cycles
CS Stall Rate				pct
Scratch Stall	0	0	0	Cycles
Insufficient SIMD Waveslots	52,228,017	15,752,457	107,668,708	#SIMD
Insufficient SIMD VGPRs	0	0	0	#SIMD
Insufficient SIMD SGPRs	0	0	0	#SIMD
Insufficient CU LDS	0	0	0	#CU
Insufficient CU Barries	0	0	0	#CU
Insufficient Bulky Resource	0	0	0	#CU
Reach CU Threadgroups Limit	0	0	0	Cycles
Reach CU Wave Limit	0	0	0	Cycles

Roofline Examples on AMD Instinct™ MI210 GPU

Roofline Plot – AMD Instinct™ MI250X Accelerators

- Device: Instinct™ MI250X
  - ORNL Frontier GPU
- Instinct™ MI250X (Dual GCDs)
- Figure shows single GCD
- Methodology applies to all AMD Instinct™ MI200 series GPUs



*AMD Instinct™ MI250X accelerator Datasheet: amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf

Roofline Example #1 – Add

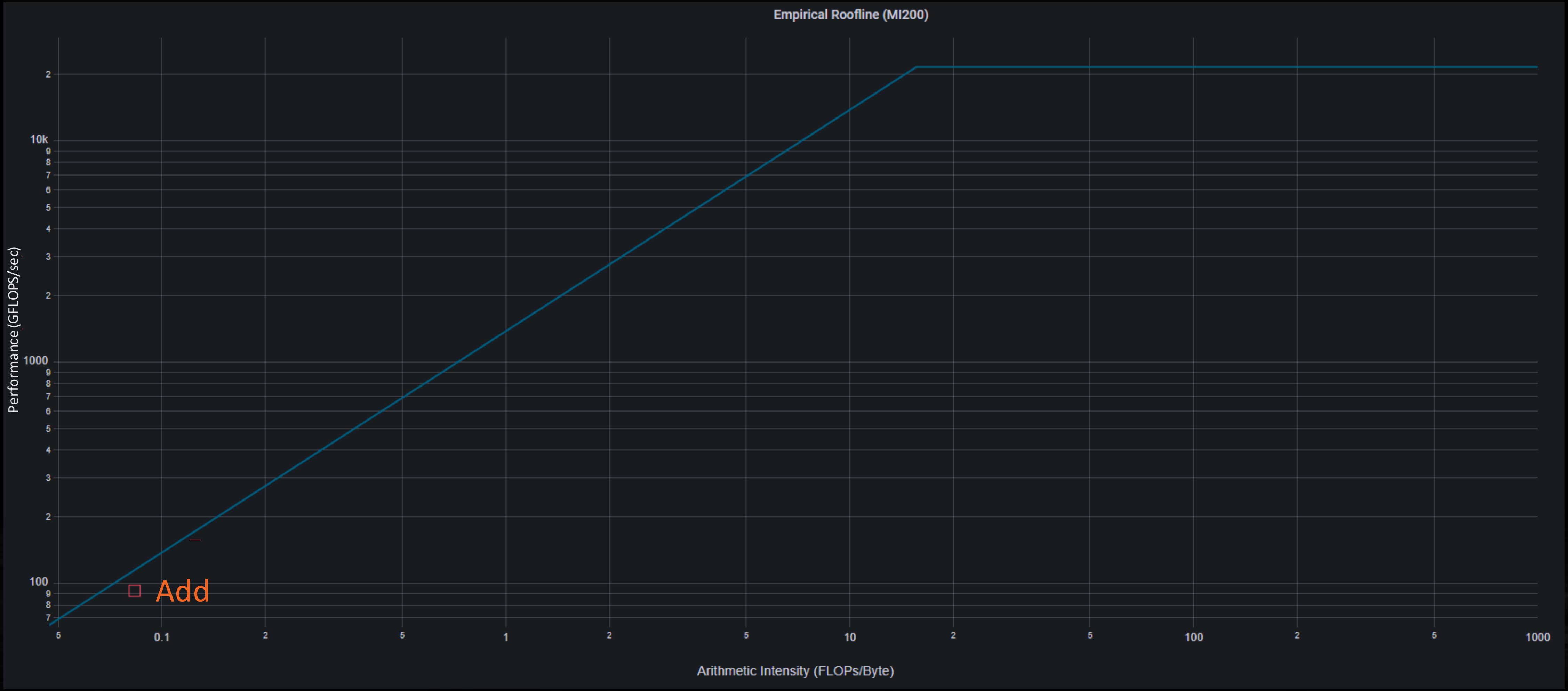
- Calculation:
  - a[i] = a[i] + b[i]
- VALU Ops Per Thread:
  - 1x V_ADD
- HBM MEM Ops Per Thread:
  - 2x RD
  - 1x WR
- Arithmetic Intensity:
  - 1 FLOP / (3 * 4Byte) = 1/12

```
template<typename T>
__global__ void add_benchmark(T *buf1, T *buf2, uint32_t nSize)
{
    const uint32_t gid = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    const uint32_t nThreads  = gridDim.x * blockDim.x;


    T *a, *b;
    a = &buf1[gid];
    b = &buf2[gid];


    for(uint32_t offset=0; offset < nSize; offset += nThreads)
    {
      a[offset] = a[offset] + b[offset];
    }
}
```

Roofline Example #1 – Add

- Calculation:
  - a[i] = a[i] + b[i]

- Reading two floats for every add results in low arithmetic intensity and HBM limited



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57

Roofline Example #2 – Mul

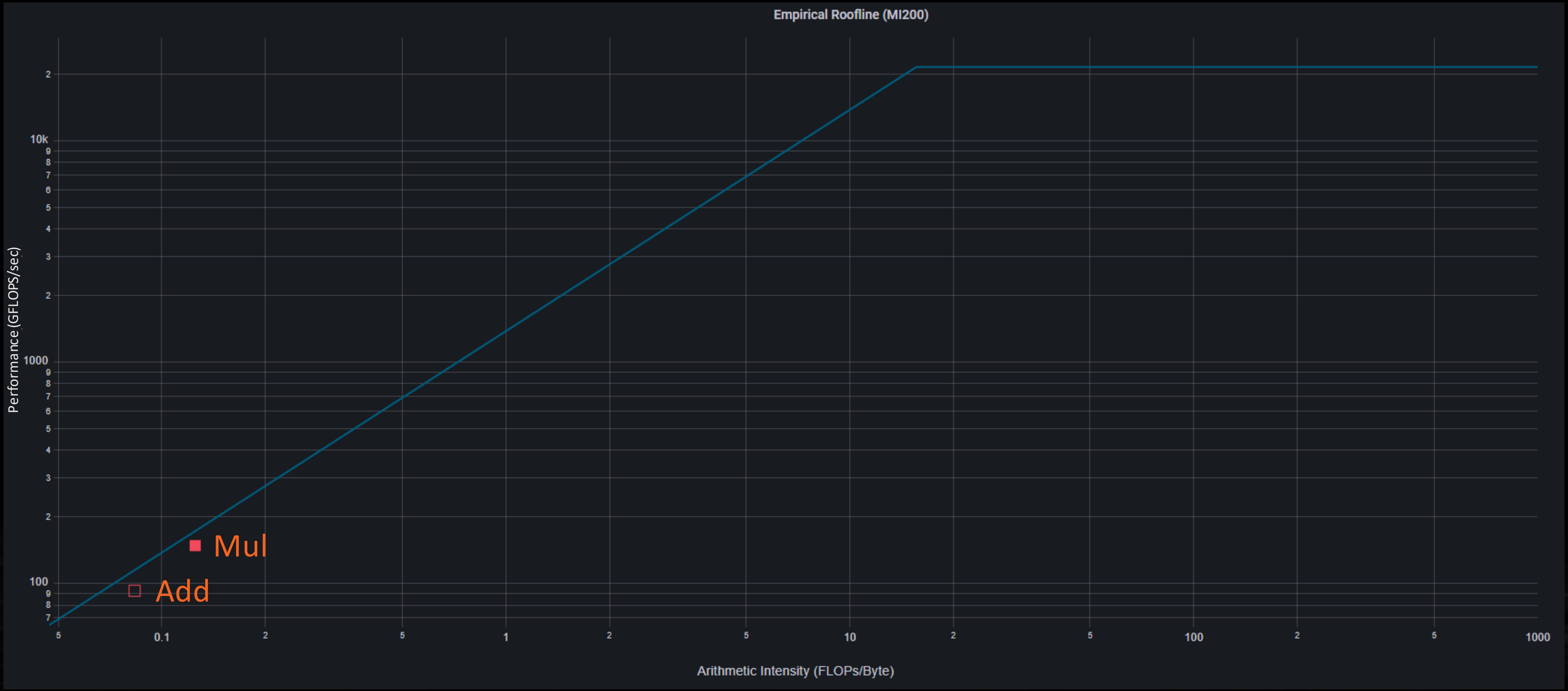
- Calculation:
  - a[i] = x * b[i]
- VALU Ops Per Thread:
  - 1x V_MUL
- HBM MEM Ops Per Thread:
  - 1x RD
  - 1x WR
- Arithmetic Intensity:
  - 1 FLOP / (2 * 4Byte) = 1/8

```
template<typename T>
__global__ void mul_benchmark(T *buf1, T *buf2, uint32_t nSize)
{
    const uint32_t gid = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    const uint32_t nThreads  = gridDim.x * blockDim.x;


    T *a, *b;
    a = &buf1[gid];
    b = &buf2[gid];
    const T x = (T)1.2;


    for(uint32_t offset=0; offset < nSize; offset += nThreads)
    {
        a[offset] = x * b[offset];
    }
}
```

Roofline Example #2 – Mul

- Calculation:
  - a[i] = c * b[i]

- Reading one less float (compared to Add) increases our arithmetic intensity and reduces sensitivity to HBM



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57, MI200-58

Roofline Example #3 – Triad

- Calculation:
  - a[i] = b[i] + x * a[i]
- VALU Ops Per Thread:
  - 1x V_ADD
  - 1x V_MUL

  (1x V_ADD + 1x V_MUL → 1x V_FMA)
- HBM MEM Ops Per Thread:
  - 2x RD
  - 1x WR
- Arithmetic Intensity:
  - 2 FLOP / (3 * 4Byte) = 1/6

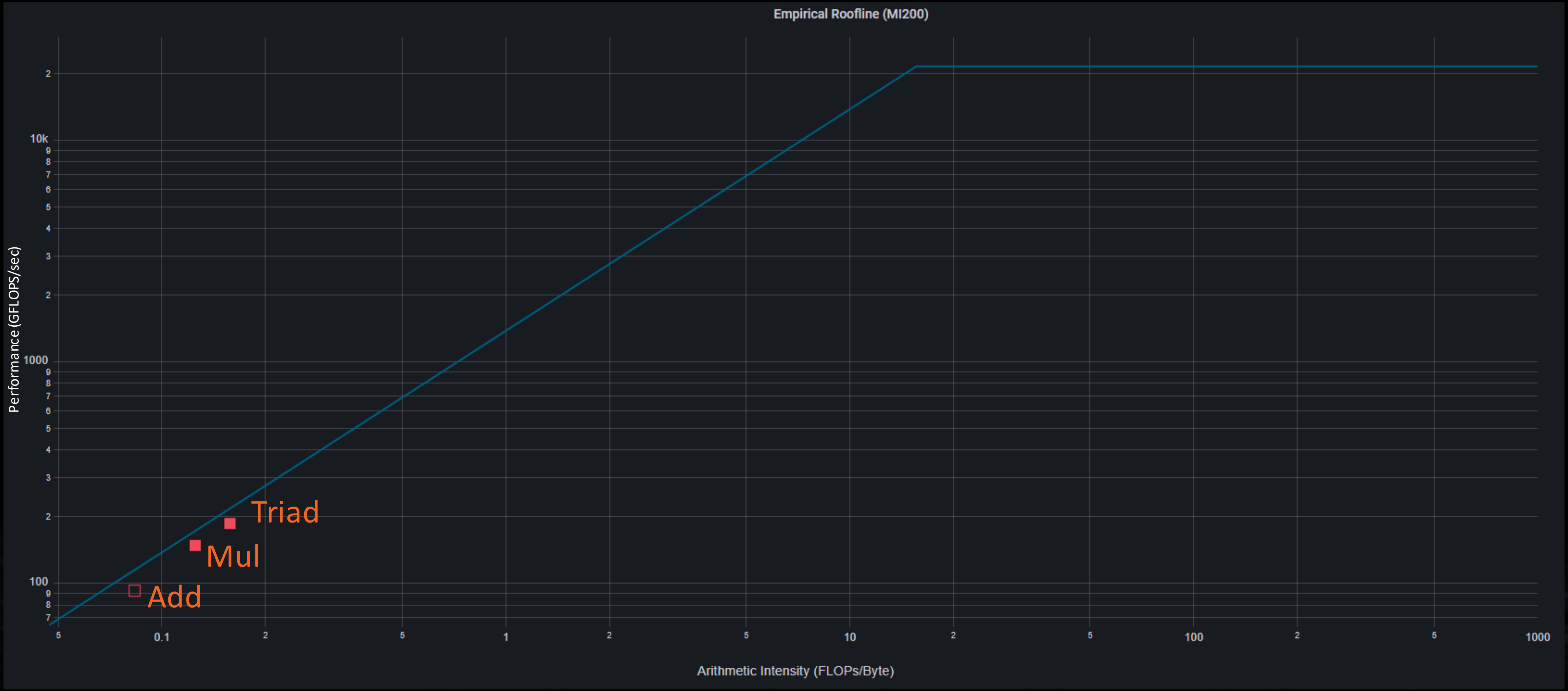
```
template<typename T>
__global__ void triad_benchmark(T *buf1, T *buf2, uint32_t nSize)
{
    const uint32_t gid = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    const uint32_t nThreads  = gridDim.x * blockDim.x;


    T *a, *b;
    a = &buf1[gid];
    b = &buf2[gid];
    const T x = (T)1.2;


    for(uint32_t offset=0; offset < nSize; offset += nThreads)
    {
        a[offset] = b[offset] + x * a[offset];
    }
}
```


AMD

Roofline Example #3 – Triad

- Calculation:
  - a[i] = b[i] + x * a[i]

- Performing an extra operation increases arithmetic intensity and further reduces sensitivity to HBM as compared to Add and Mul



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57, MI200-58, MI200-59

Roofline Example #4 – FMA

- Calculation:
  - x = a[i] * x + y
- VALU Ops Per Thread:
  - 1x V_ADD
  - 1x V_MUL

  1x V_FMA
- HBM MEM Ops Per Thread:
  - 1x RD
- Arithmetic Intensity:
  - 2 FLOP / (1 * 4Byte) = 1/2

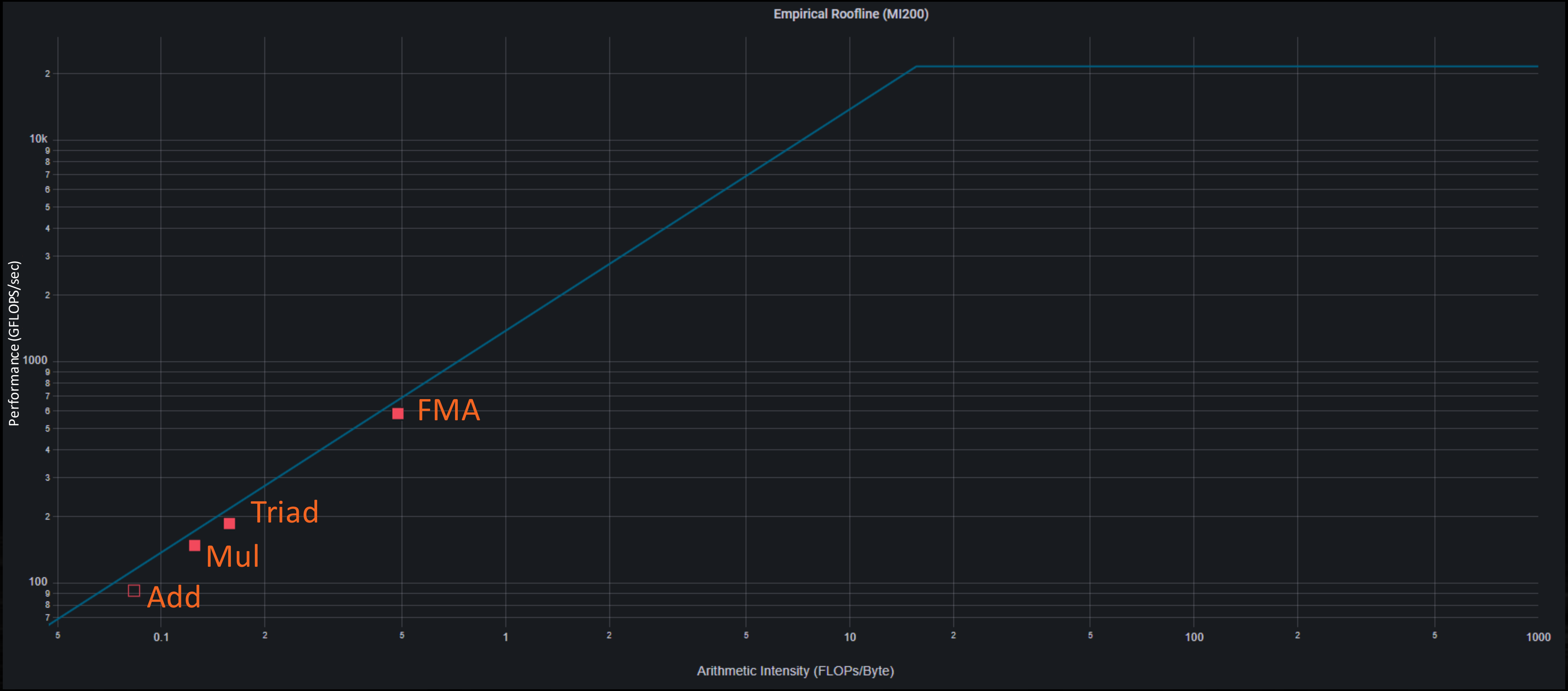
```
template<typename t>
__global__ void flops_benchmark(T *buf, uint32_t nSize)
{
    const uint32_t gid = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    const uint32_t nThreads  = gridDim.x * blockDim.x;


    T *a;
    a = &buf[gid];
    const T y = (T) 1.0;
    T x = (T) 2.0;


    for(uint32_t offset=0; offset < nSize; offset += nThreads)
    {
        x = a[offset] * x + y;
    }
    a[0] = -x;
}
```


AMD

Roofline Example #4 – FMA

- Calculation:
  - x = a[i] * x + y

- Each thread having to load one less value from HBM further increases arithmetic intensity and improves FLOPs/s performance



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57, MI200-58, MI200-59, MI200-60

Roofline Example #5 – FMA 1024

- Calculation:
  - x = a[i] * x + y
- VALU Ops Per Thread:
  - 1x V_ADD
  - 1x V_MUL

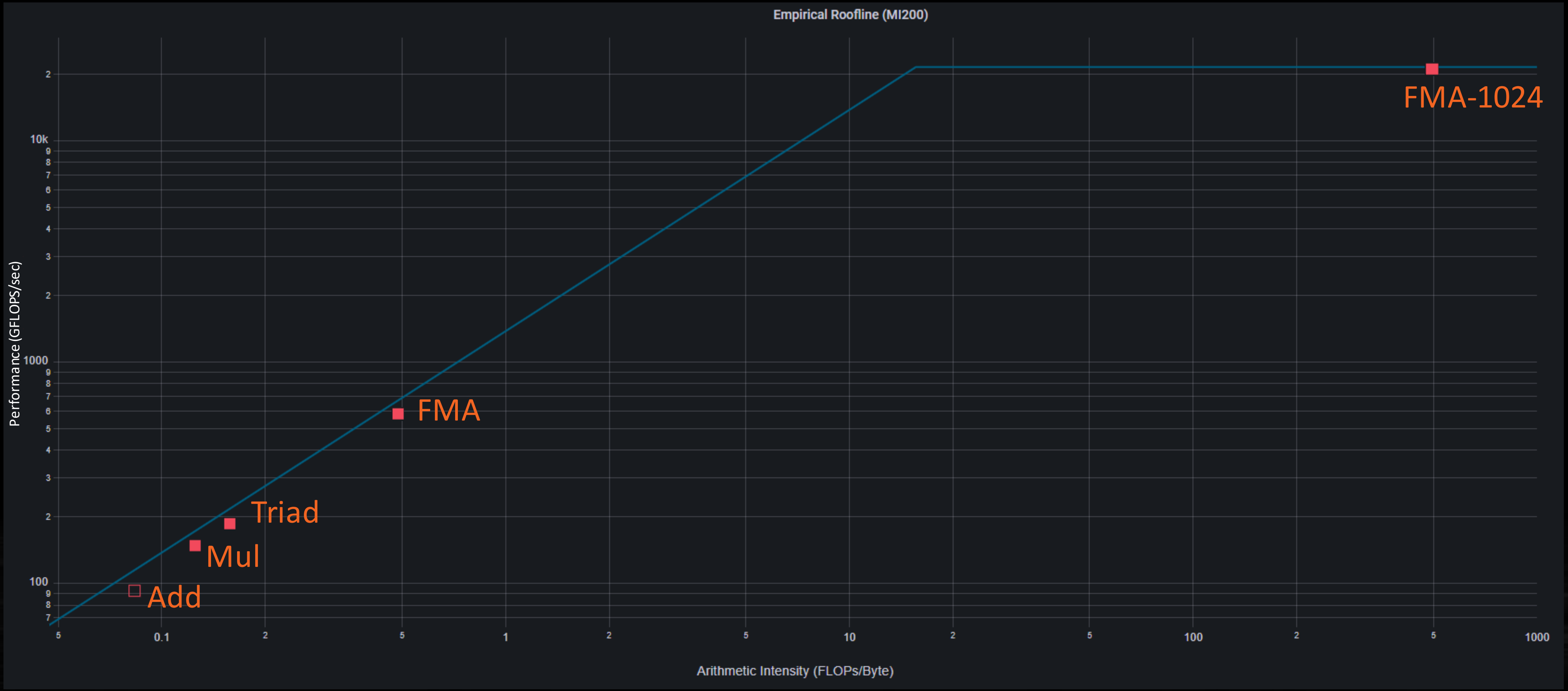
  1x V_FMA
- HBM MEM Ops Per Thread:
  - 1x RD
- Arithmetic Intensity:
  - 1024 * 2 FLOP / (1 * 4Byte) = 512

1024

```
template<typename T, int nFMA>
__global__ void flops_benchmark(T *buf, uint32_t nSize)
{
    const uint32_t gid = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
    const uint32_t nThreads  = gridDim.x * blockDim.x;


    T *a;
    a = &buf[gid];
    const T y = (T) 1.0;
    T x = (T) 2.0;


    for(uint32_t offset=0; offset < nSize; offset += nThreads)
    {
        for(int j=0; j<nFMA; j++)
        {
            x = a[offset] * x + y;
        }
    }
    a[0] = -x;
}
```

Roofline Example #5 – FMA 1024

- Calculation:
  - x = a[i] * x + y
- Each thread looping over many FMAs with only one read significantly increases arithmetic intensity and becomes compute VALU limited

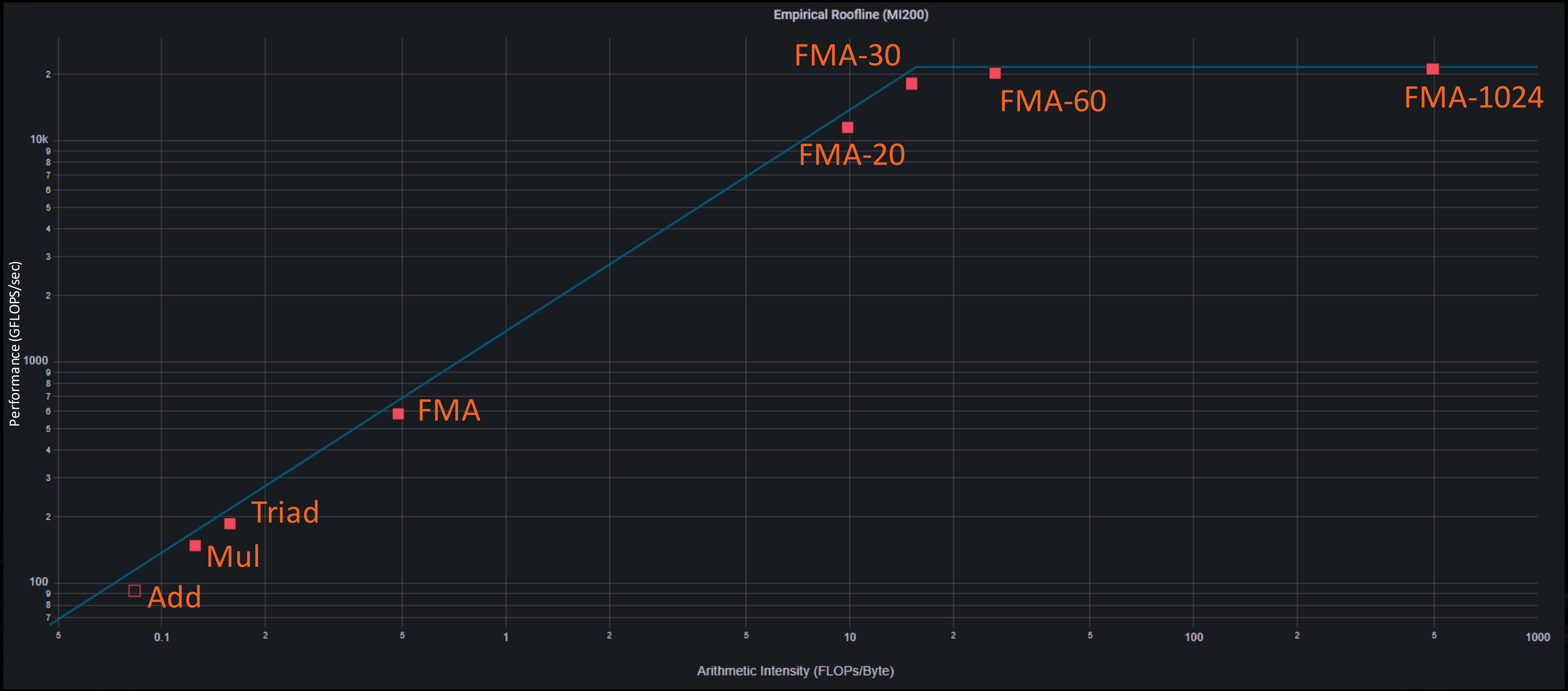


*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57, MI200-58, MI200-59, MI200-60

Roofline Example #6 – FMA Sweep

- Calculation:
  - x = a[i] * x + y

- Further sweeping the number of FMA instructions from 20 to 60 shows the workload transitioning from HBM limited to VALU limited

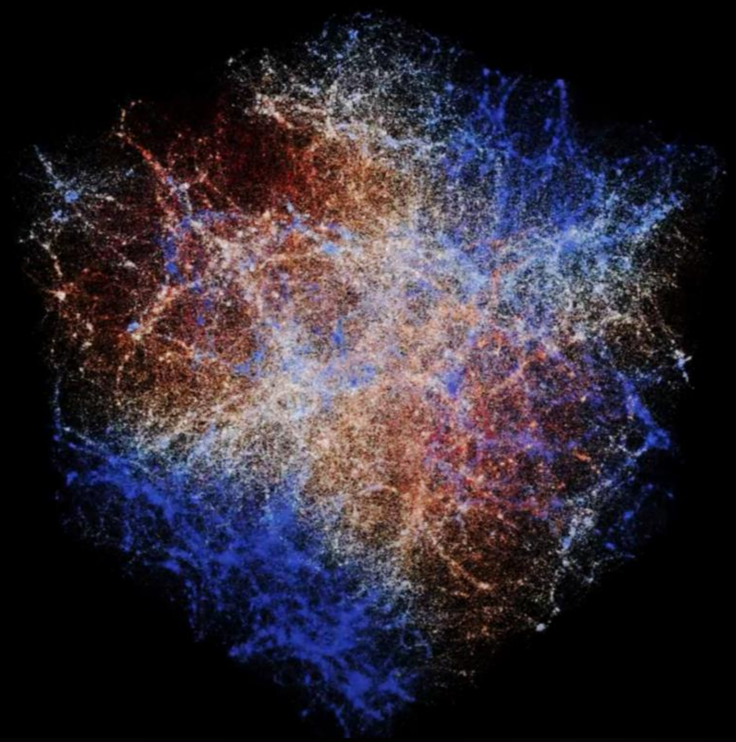


*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-57, MI200-58, MI200-59, MI200-60

Roofline Use Case - HPC Particle Codes

- Particle interactions form the foundation of many computational science codes from multiple domains
  - Domains: Cosmology, astrophysics, molecular dynamics, and more
  - Applications: HACC, LAMMPS, NAMD, Amber, GROMACS

HACC – Cosmology



Nbody

- One such computational algorithm for computing particle interactions leveraged by these applications
- Direct particle-particle method
- Highly accurate
- Computationally expensive (N^2)

Roofline Example #1 – Nbody

- Repo: https://github.com/ROCm-Developer-Tools/HIP-Examples/tree/master/mini-nbody/hip
  - Fundamental particle-particle algorithm
  - Single collection of N particles calculating N^2 pair-wise interactions
  - Double precision (FP64)
  - Multiple implementations leveraging different optimization approaches
- "orig"
  - Numerical Computing 101 unoptimized implementation
- "soa"
  - Converting particle data layout from array of structures to structure of arrays
- "block"
  - Loading and computing particle data in "tiles" to increase cache hits
- "unroll"
  - Adding #pragma unroll to particle "tile" processing for loop

Roofline Example #1 – Nbody

- “orig”
  - Numerical Computing 101 unoptimized implementation
- $O(n^2)$ Interaction Ops:
  - 3x V_ADD
  - 6x V_FMA
  - 2x V_MUL
  - 1x V_DIV
  - 1x V_SQRT
  - 3x RD

V_RSQ

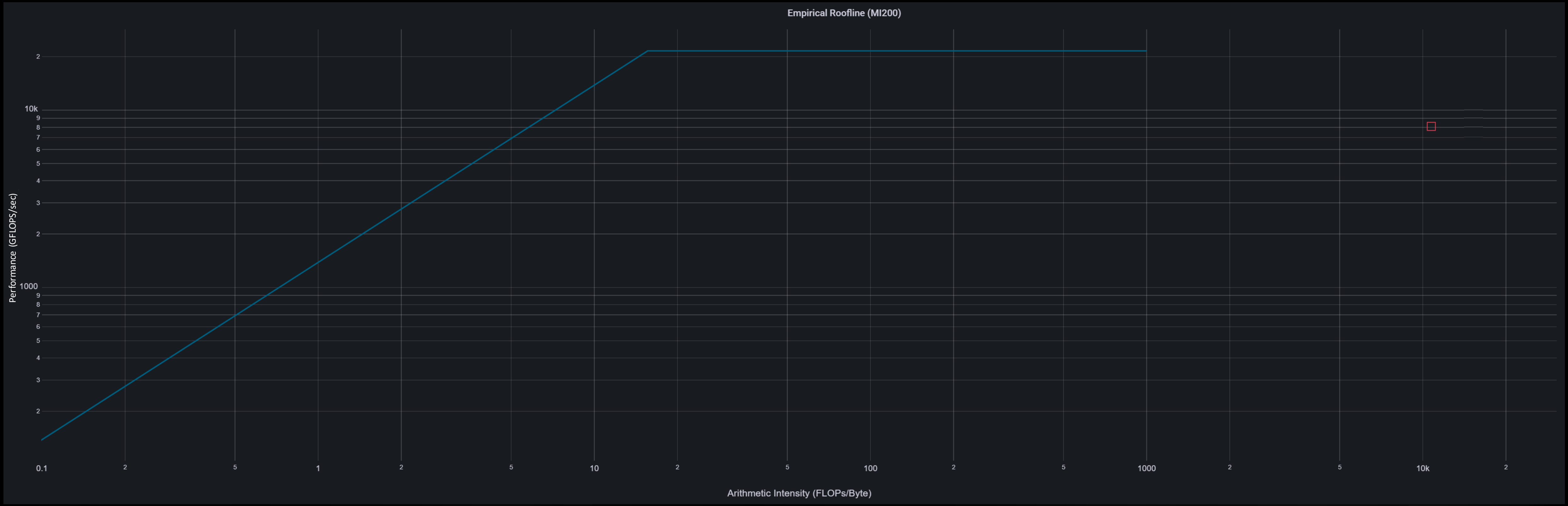
- O(n) Accumulation Ops:
  - 3x V_FMA
  - 3x RD
  - 3x WR
- Interaction AI:
  - $[(3 + 12 + 2 + 1)\text{FLOPs} / 24\text{Bytes}] * n^2 = (3/4)n^2$
- Accumulation AI:
  - $(6 \text{ FLOPs} / 24 \text{ Bytes}) * n = n/4$

```
typedef struct { double x, y, z, vx, vy, vz; } Body;

__global__
void bodyForce(Body *p, double dt, int n) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < n) {
    double Fx = 0.0f; double Fy = 0.0f; double Fz = 0.0f;

    for (int j = 0; j < n; j++) {
      double dx = p[j].x - p[i].x;
      double dy = p[j].y - p[i].y;
      double dz = p[j].z - p[i].z;
      double distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      double invDist = rsqrtf(distSqr);
      double invDist3 = invDist * invDist * invDist;

      Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
    }

    p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
  }
}
```

Roofline Example #1 – Nbody

- “orig”
  - Numerical Computing 101 unoptimized implementation

- Nbody has a very high arithmetic intensity and therefore closer to the top of the roofline (compute sensitive)
- Transcendentals like RSQ do not complete at same rate as ADD, MUL and FMA and therefore limit the peak FLOPS/s performance



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-61

Roofline Example #1 – Nbody

- “block”
  - Preload a “tile” size worth of particle data into faster shared memory for computing O(n2) forces
- Processing in “tiles” improves reuse and increases cache hits

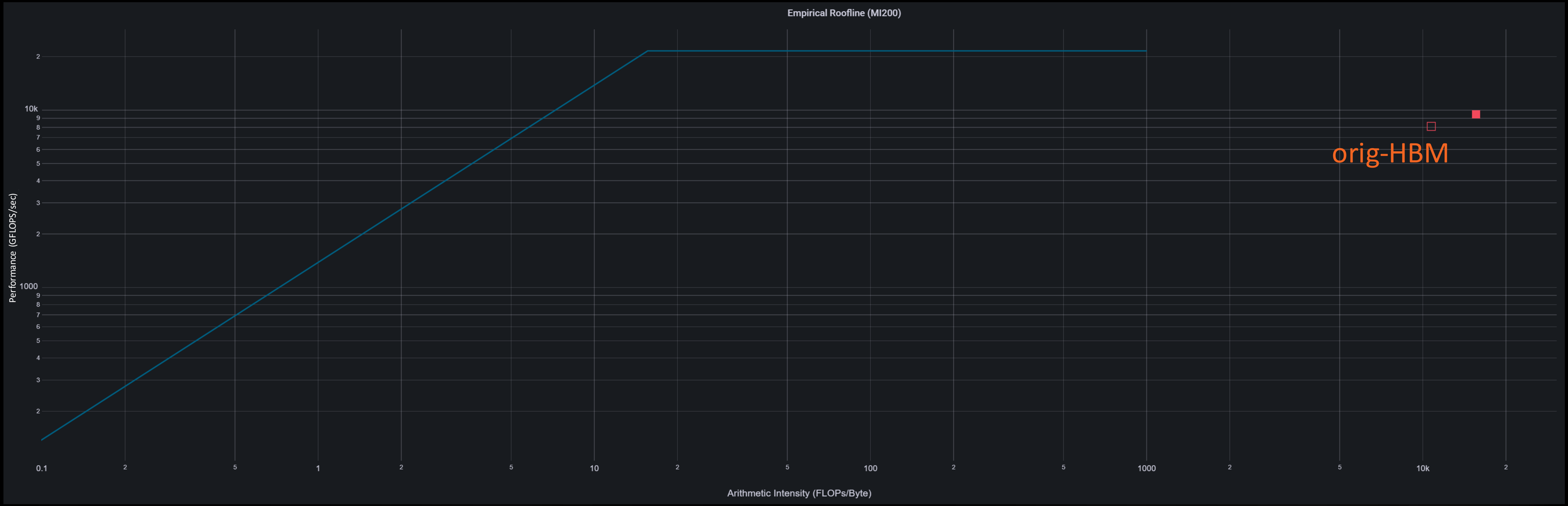
```
typedef struct { double4 *pos, *vel; } BodySystem;

__global__
void bodyForce(double4 *p, double4 *v, double dt, int n) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < n) {
    double Fx = 0.0f; double Fy = 0.0f; double Fz = 0.0f;

    for (int tile = 0; tile < gridDim.x; tile++) {
      __shared__ double3 spos[BLOCK_SIZE];
      double4 tpos = p[tile * blockDim.x + threadIdx.x];
      spos[threadIdx.x] = make_double3(tpos.x, tpos.y, tpos.z);
      __syncthreads();

      for (int j = 0; j < BLOCK_SIZE; j++) {
        double dx = spos[j].x - p[i].x;
        double dy = spos[j].y - p[i].y;
        double dz = spos[j].z - p[i].z;
        double distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
        double invDist = rsqrtf(distSqr);
        double invDist3 = invDist * invDist * invDist;

        Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
      }
      __syncthreads();
    }

    v[i].x += dt*Fx; v[i].y += dt*Fy; v[i].z += dt*Fz;
  }
}
```


AMD

Roofline Example #1 – Nbody

- "block"
  - Loading and computing particle data in "tiles" to increase cache hits

- Working on smaller "tiles" of particles improves cache hits, removing loads from HBM and increasing FLOPs performance



*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-62

Roofline – All Workloads

Orange: Synthetic Workload Yellow: Proxy app Green: Full app

*MI250x. RESULTS MAY VARY. SEE ENDNOTE: MI200-62

DISCLAIMERS AND ATTRIBUTIONS

NAMD was developed by the Theoretical Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. http://www.ks.uiuc.edu/Research/namd/

Math Kernel Endnotes

MI200-57 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of 92.6 GFLOPS/s on Add Kernel. Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-57

MI200-58 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of 149.8 GFLOPS/s on Mul Kernel. Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-58

MI200-59 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of 184.7 GFLOPS/s on Triad Kernel. Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-59

MI200-60 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of up to 21.7 TFLOPS/s on FMA Kernel. Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-60

Nbody Endnotes

MI200-61 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of 8.7 TFLOPS/s on benchmark mini-nbody-orig. Information on mini-nbody-orig: https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/mini-nbody/hip/nbody-orig.cpp. Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-61

MI200-62 - Testing Conducted by AMD performance lab as of 4/19/2022 on a single socket optimized 3rd Gen AMD EPYC™ CPU powered server with 1x AMD Instinct™ MI250X OAM (128 GB HBM2e) 560W GPU with AMD Infinity Fabric™ technology resulted in a median score of 9.5 TFLOPS/s on benchmark mini-nbody-block. Information on mini-nbody-block: https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/mini-nbody/hip/nbody-block.cpp. . Server manufacturers may vary configurations, yielding different results. Performance may vary based on use of latest drivers and optimizations MI200-62

Q&A