Introduction to Omniperf

and Hierarchical Roofline on AMD Instinct™ MI200 GPUs

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

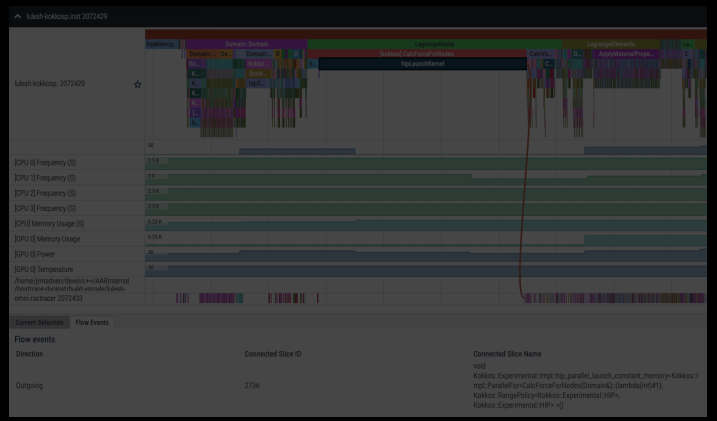
Omnitrace

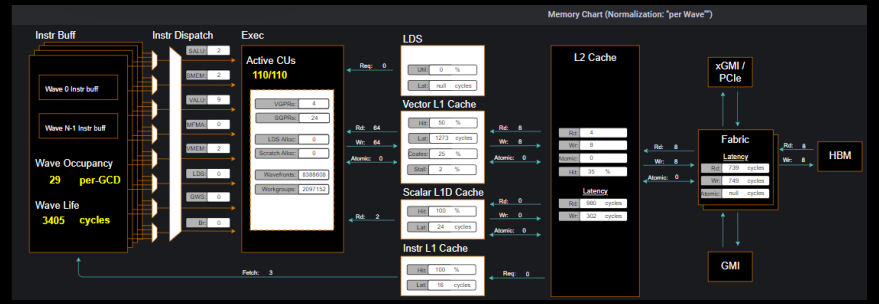
Trace collection	Comprehensive trace collection
	CPU
	GPU
Supports	CPU copy
	HIP API
	HSA API
	GPU Kernels
	OpenMP®
	MPI
	Kokkos
	p-threads
	multi-GPU
Visualisation	Traces visualized with Perfetto

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

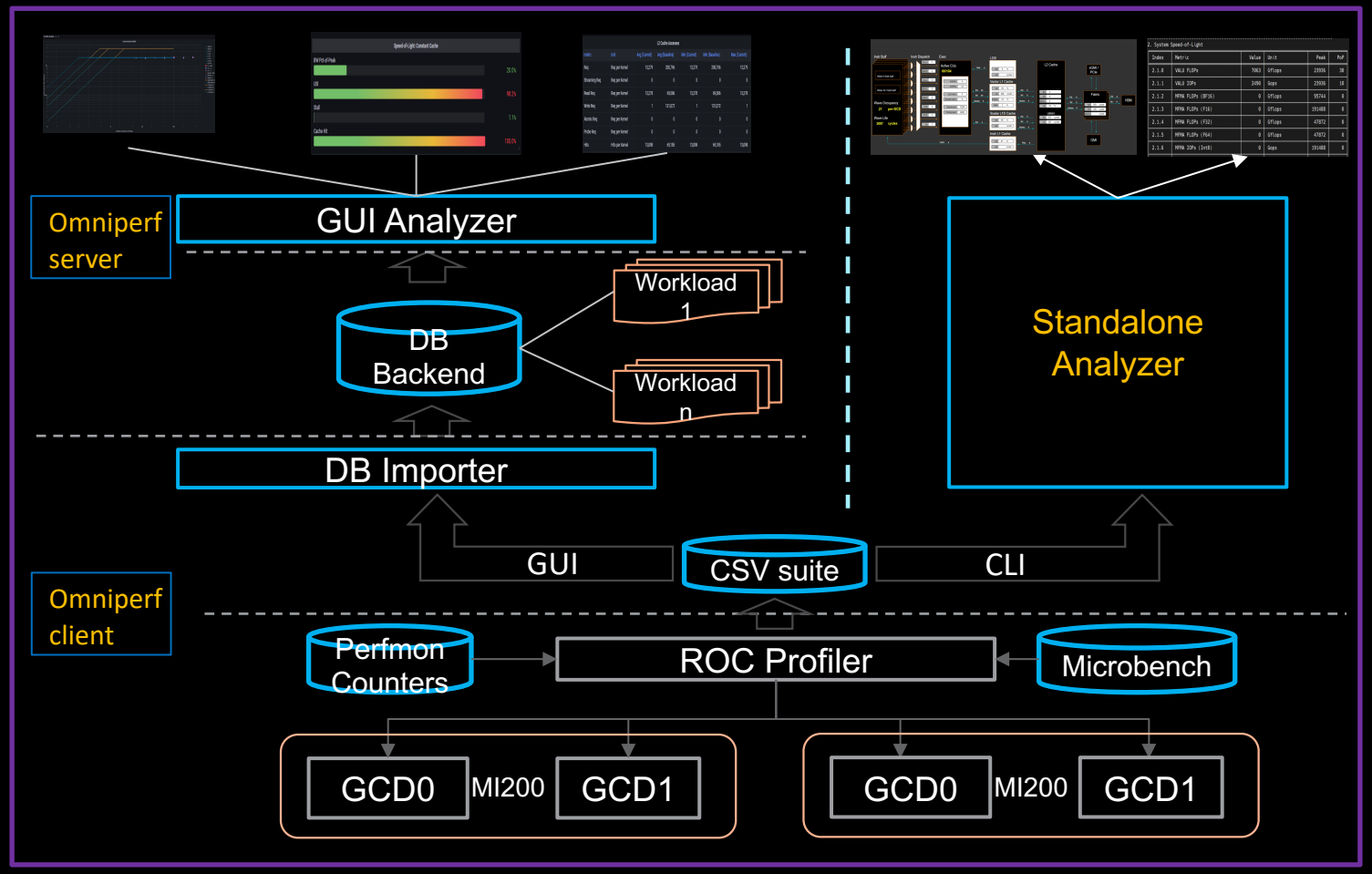
Name	Calls	TotalDura	AverageN	Percentage
hipMemcpyAsync	99	3.22E+10	3.25E+08	44.14872
hipEventSynchronize	330	2.42E+10	73394557	33.225
hipMemsetAsync	87	7.76E+09	89232696	10.64953
hipHostMalloc	9	5.41E+09	6.01E+08	7.415198
hipDeviceSynchronize	28	1.32E+09	47006288	1.805515
hipHostFree	17	1.05E+09	61534688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





Omniperf: Automated Collection of Hardware Counters and Analysis

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



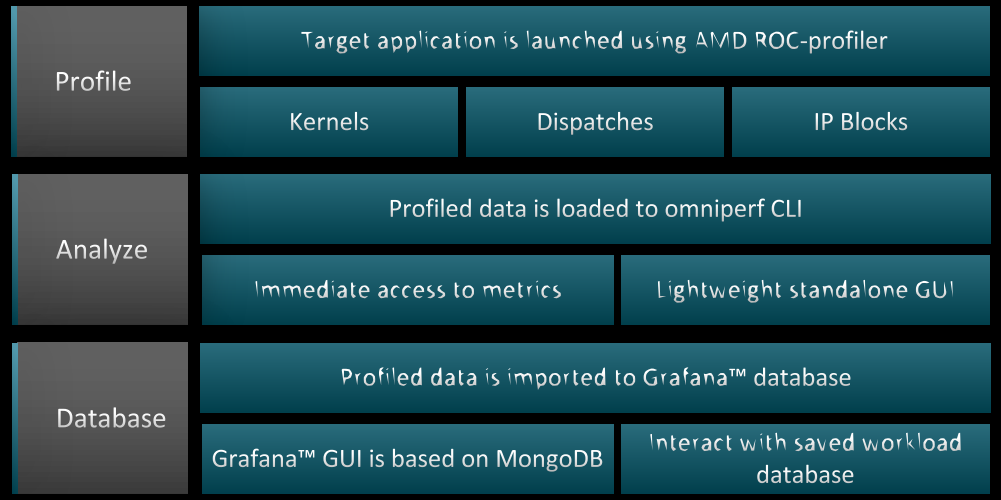
Refer to current documentation for recent updates

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

Omniperf modes



Basic command-line syntax:

Profile:

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

Analyze:

```
$ omniperf analyze -p
<path/to/workloads/workload_name/MI200/>
```

To use a lightweight standalone GUI with CLI analyzer:

```
$ omniperf analyze -p
<path/to/workloads/workload_name/MI200/> --gui
```

Database:

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

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

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

For problems, create an issue here: https://github.com/AMDResearch/omniperf/issues
Documentation: https://amdresearch.github.io/omniperf

Omniperf profiling

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

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

Compile with hipcc:

```
$ hipcc –o vcopy vcopy.cpp
```

Profile with Omniperf:

```
$ omniperf profile –n vcopy_all -- ./vcopy –n 1048576 –b 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 – code needs to be ready to be replayed. Use kernel/dispatch filters especially when trying to collect roofline analysis.

Omniperf analyze

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

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

Compile with hipcc:

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

Profile with Omniperf:

```
$ omniperf profile –n vcopy_all -- ./vcopy –n 1048576 –b 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	100.0

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

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 with standalone GUI

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

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

Compile with hipcc:

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

Profile with Omniperf:

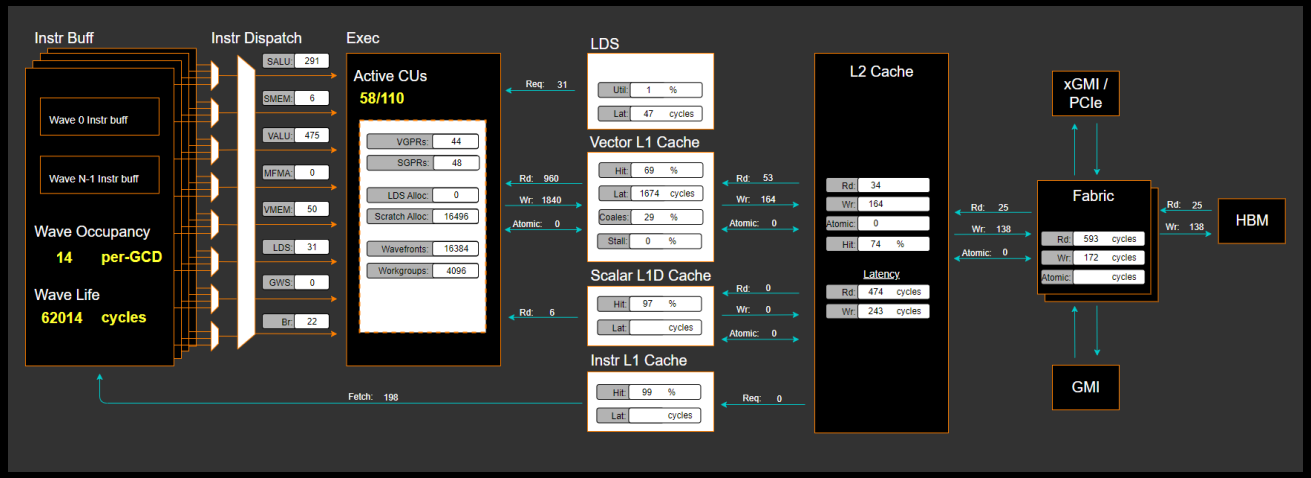
```
$ omniperf profile –n vcopy_all -- ./vcopy 1048576 256
```

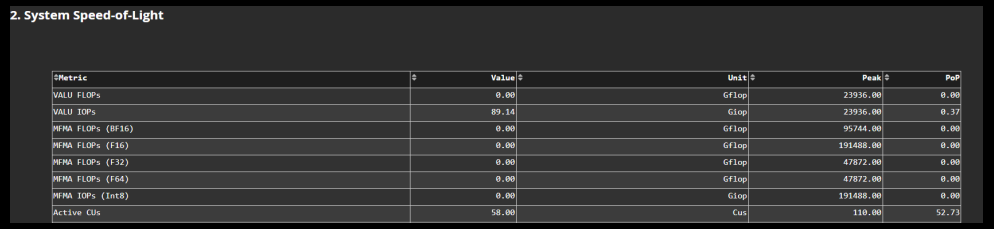
A new directory will be created called `workloads/vcopy_all`

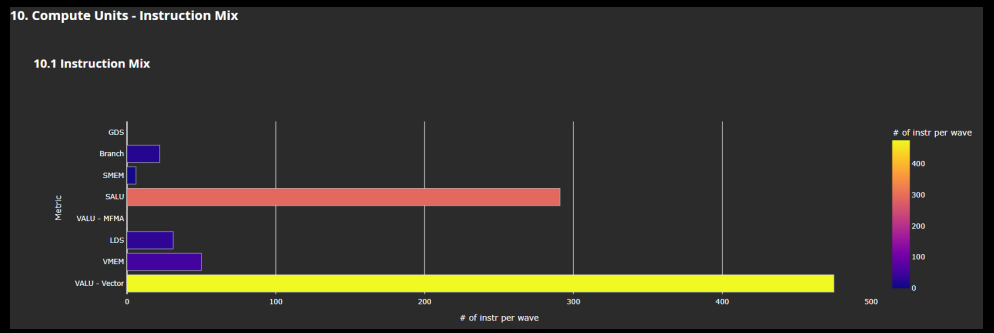
Analyze the profiled workload:

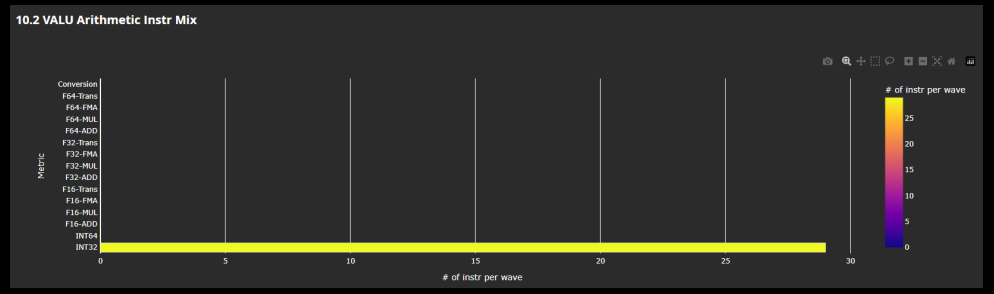
```
$ omniperf analyze –p workloads/vcopy_all/mi200/ --gui
```

Open web page http://IP:8050/









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

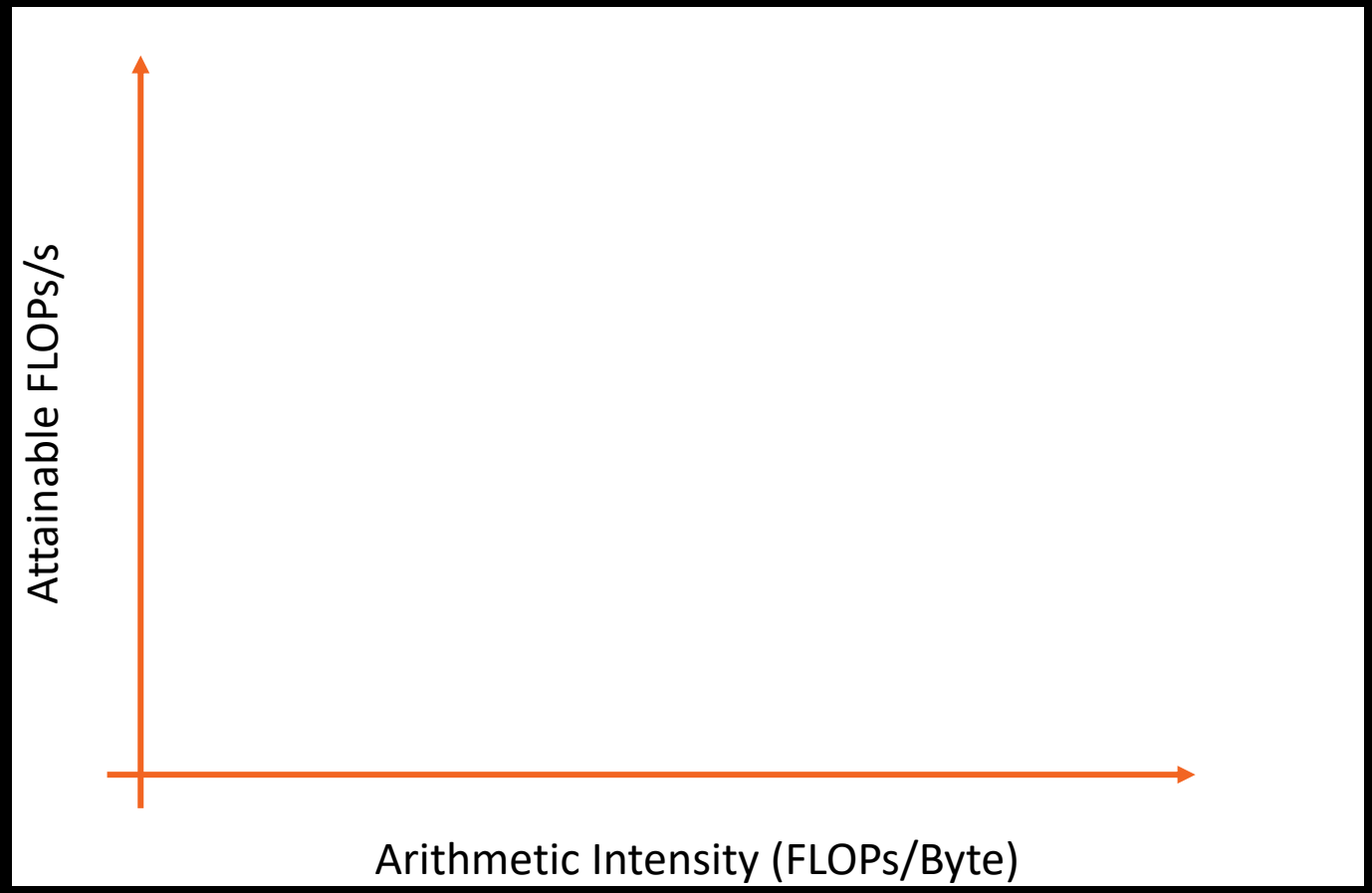
Background - What is a roofline?

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

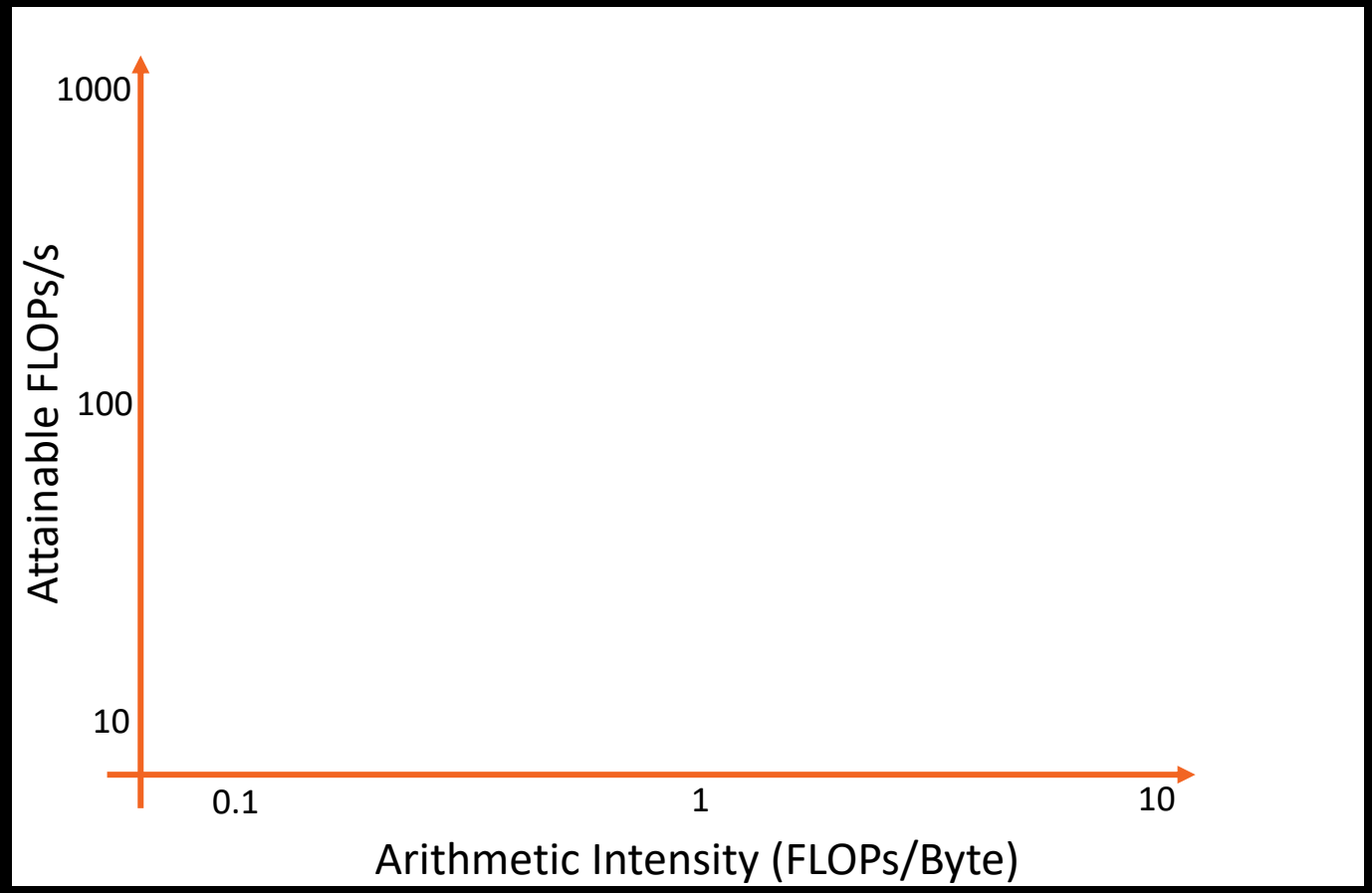
Background – What is Roofline

- 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



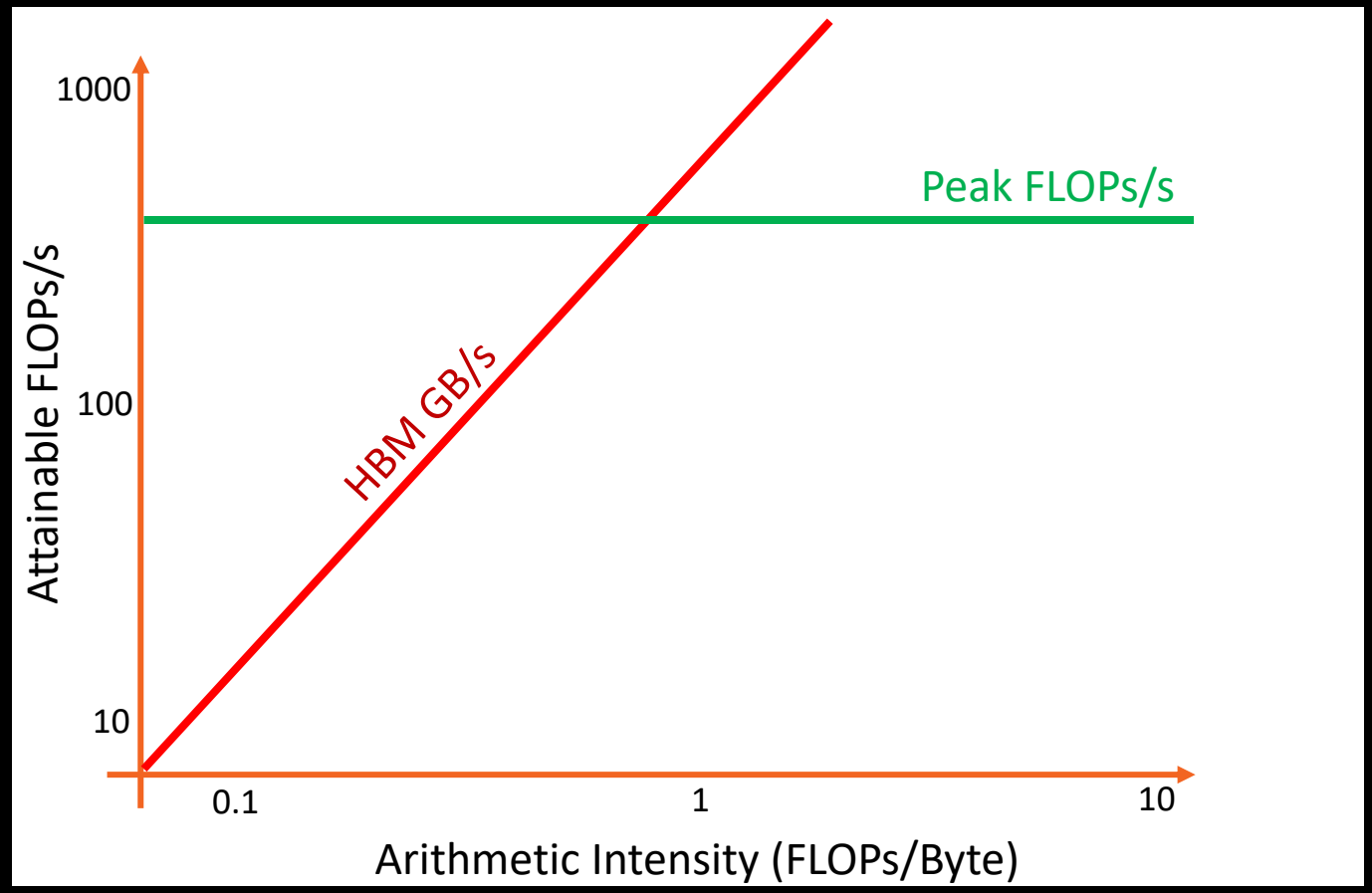
Background – What is Roofline

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



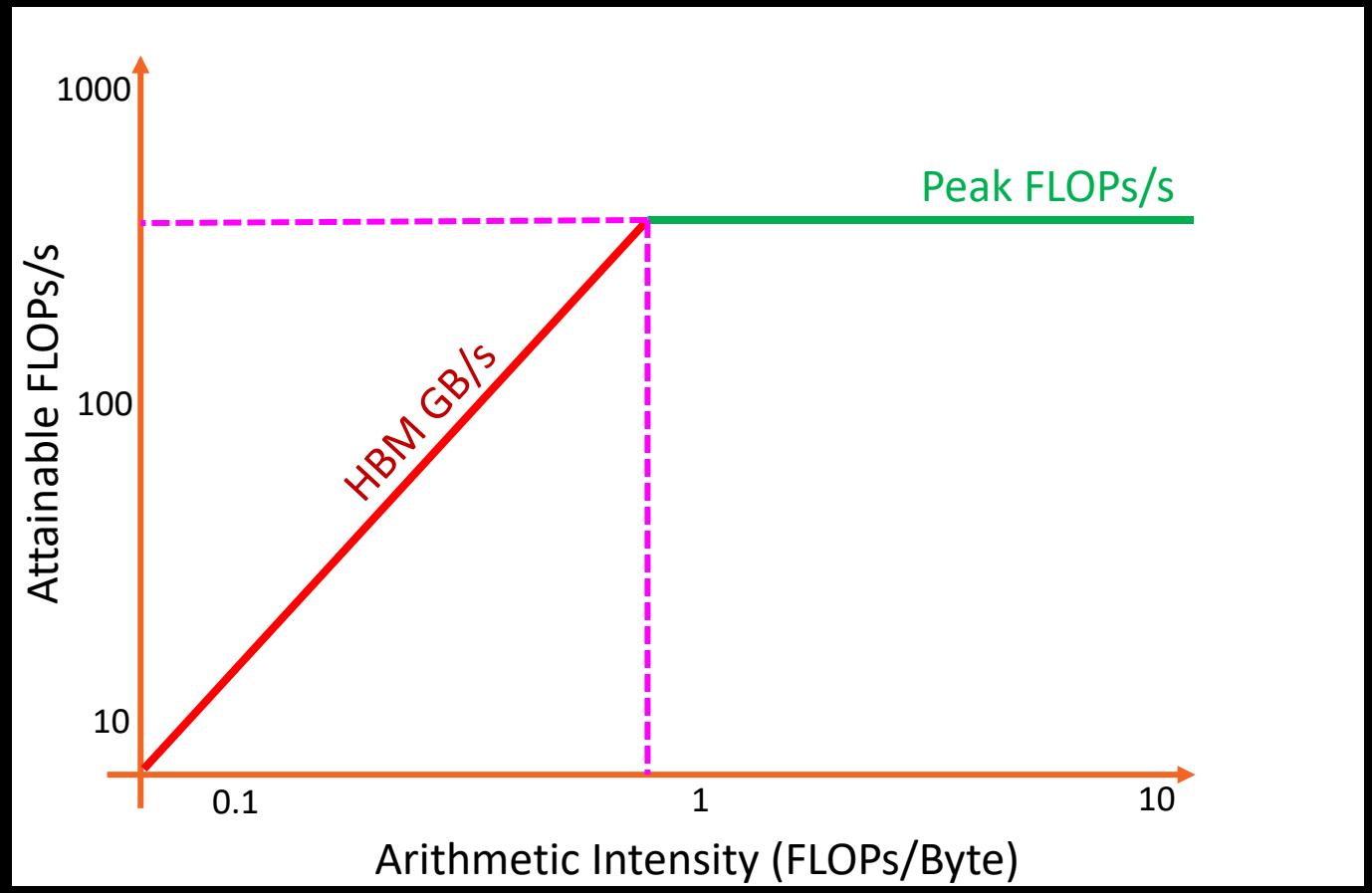
Background – What is Roofline

- 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



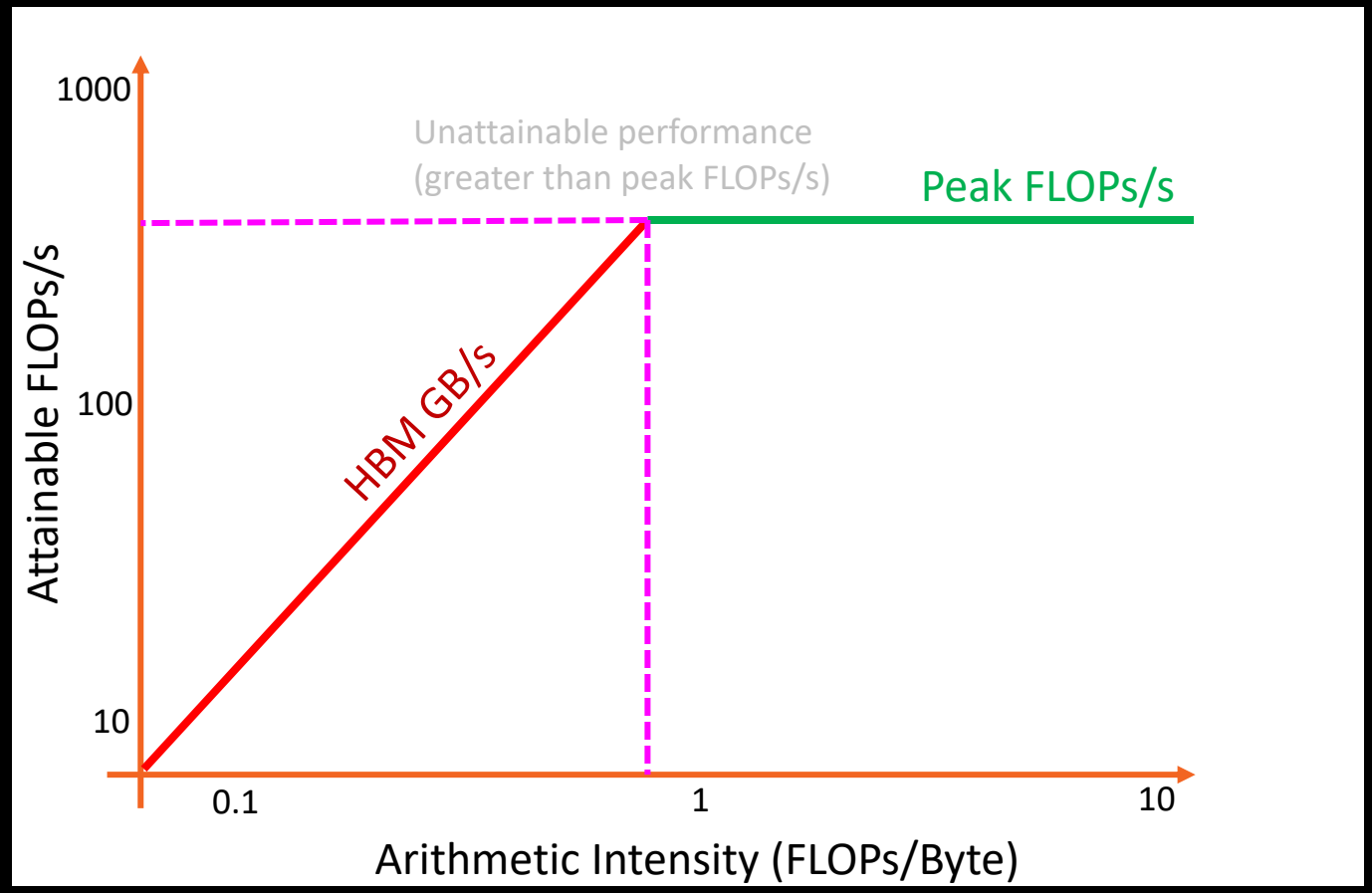
Background – What is Roofline

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



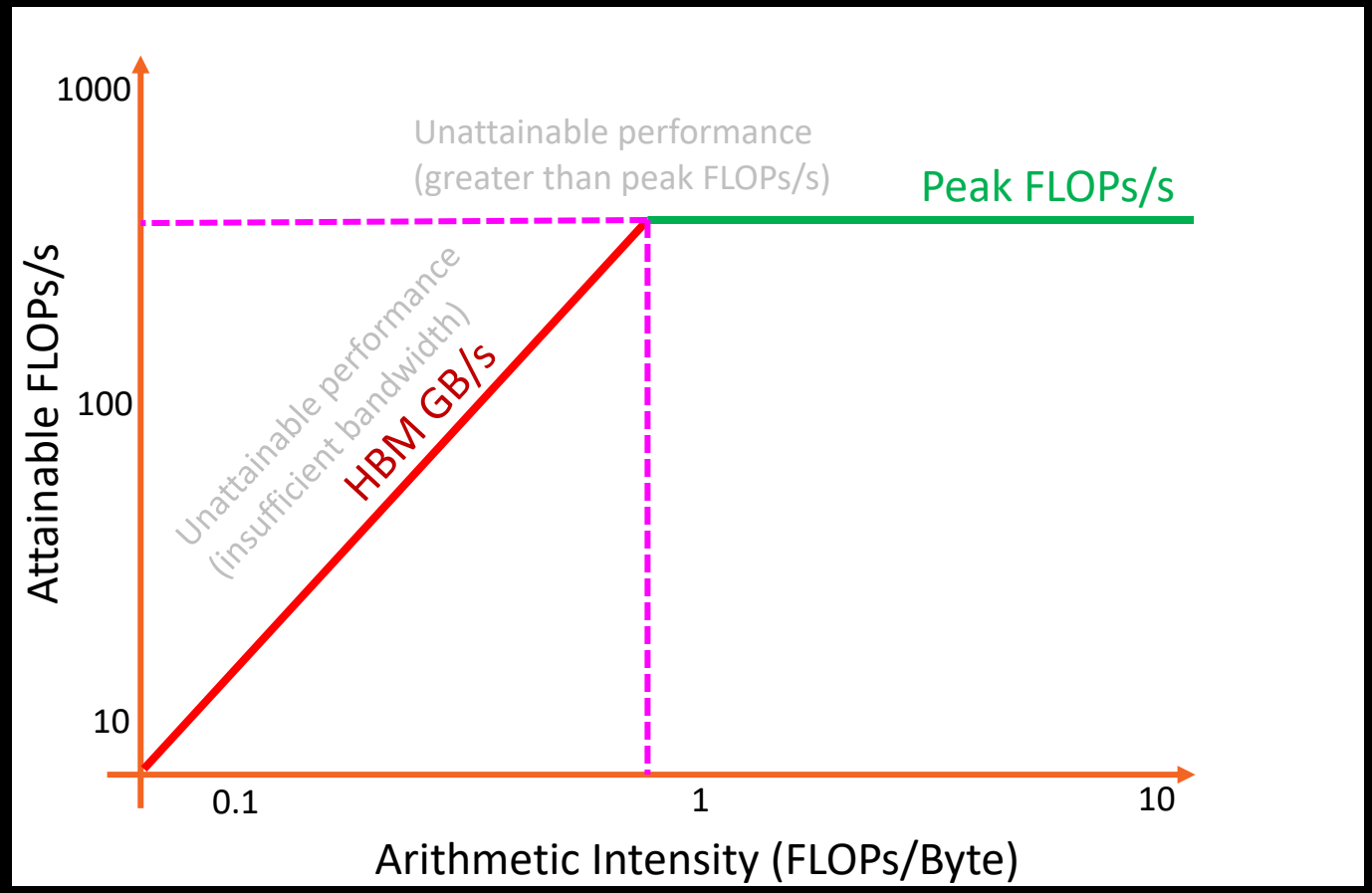
Background – What is Roofline

- 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



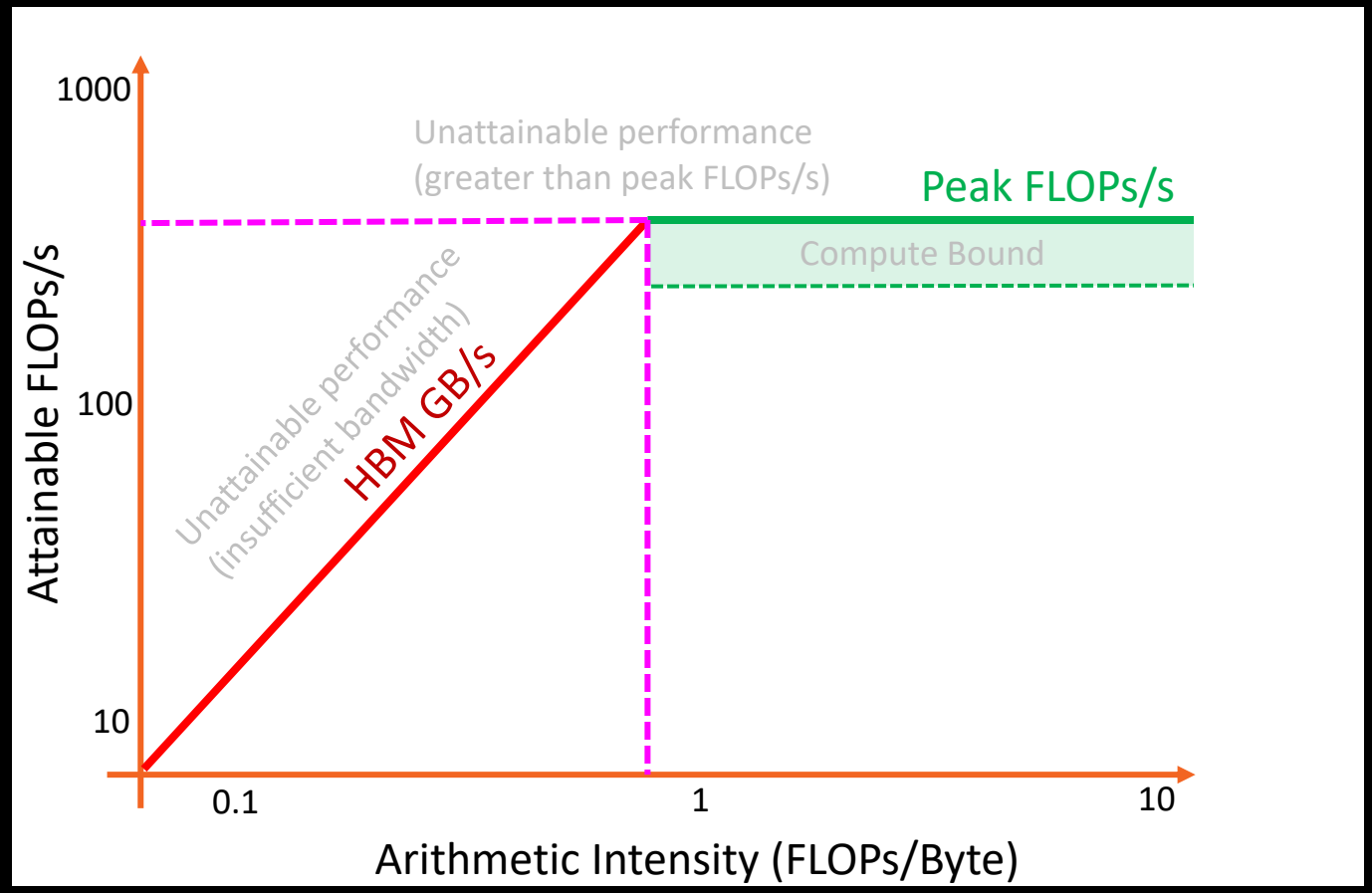
Background – What is Roofline

- 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



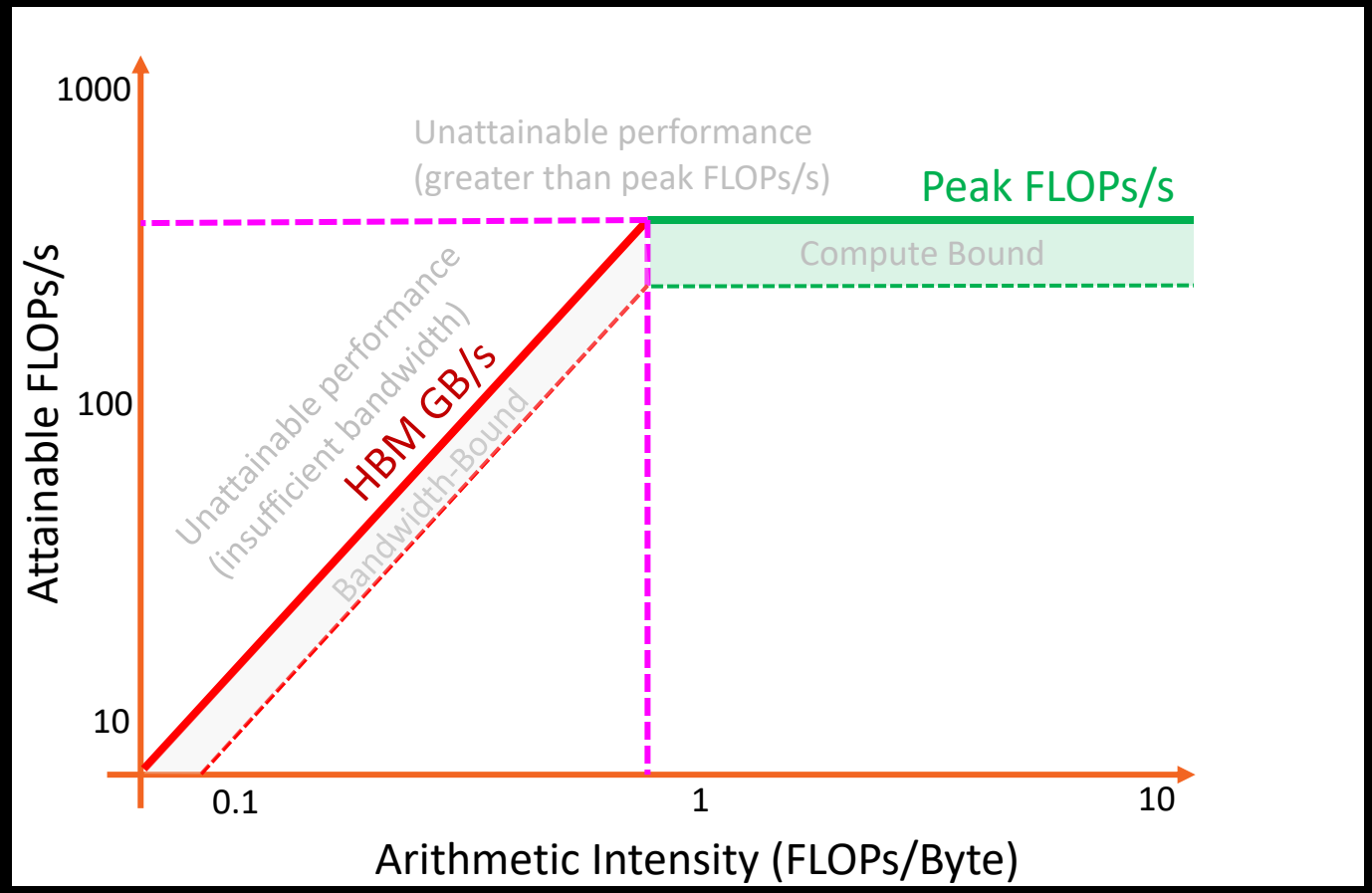
Background – What is Roofline

- 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



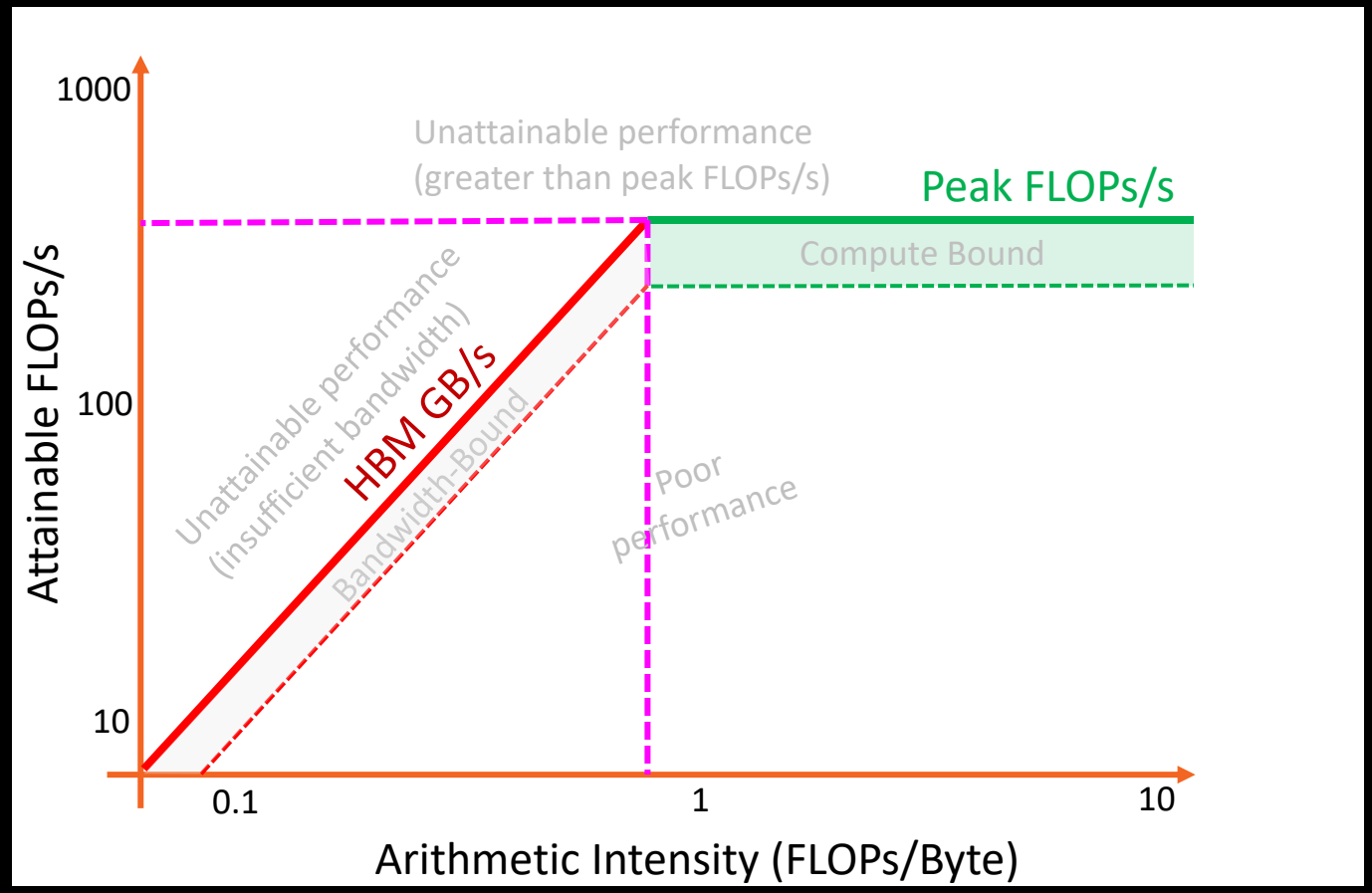
Background – What is Roofline

- 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



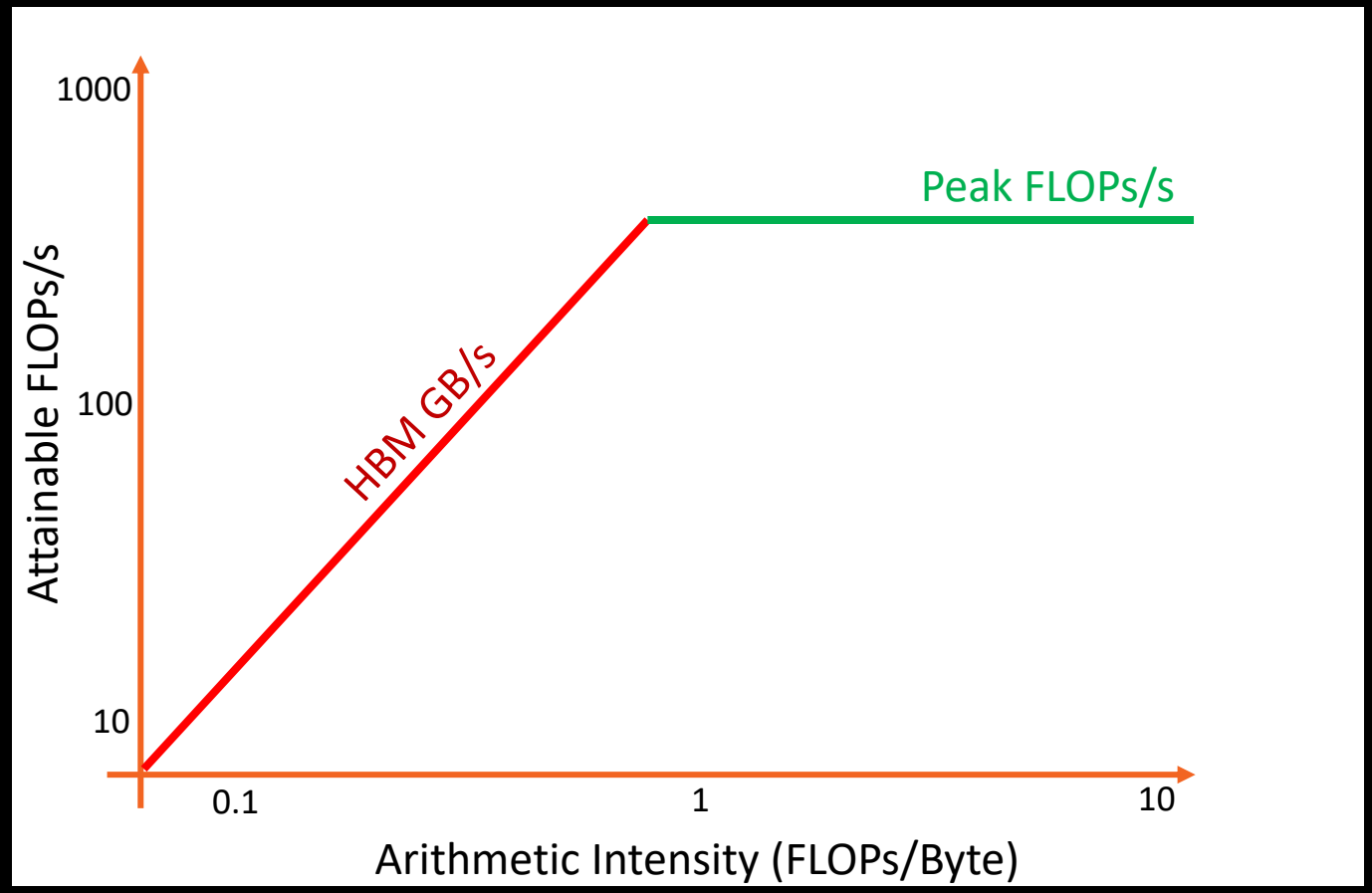
Background – What is Roofline

- 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



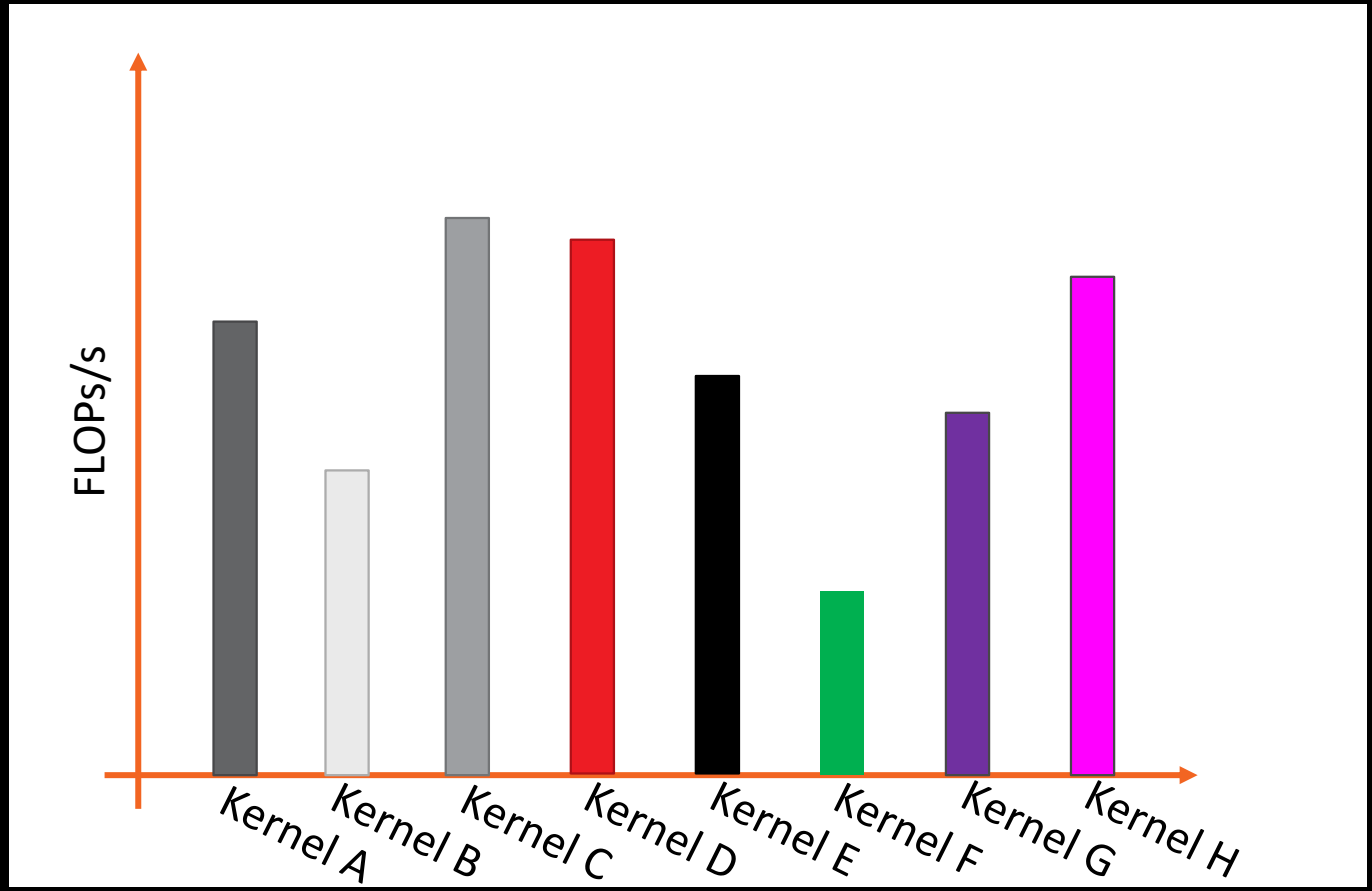
Background – What is Roofline

- 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



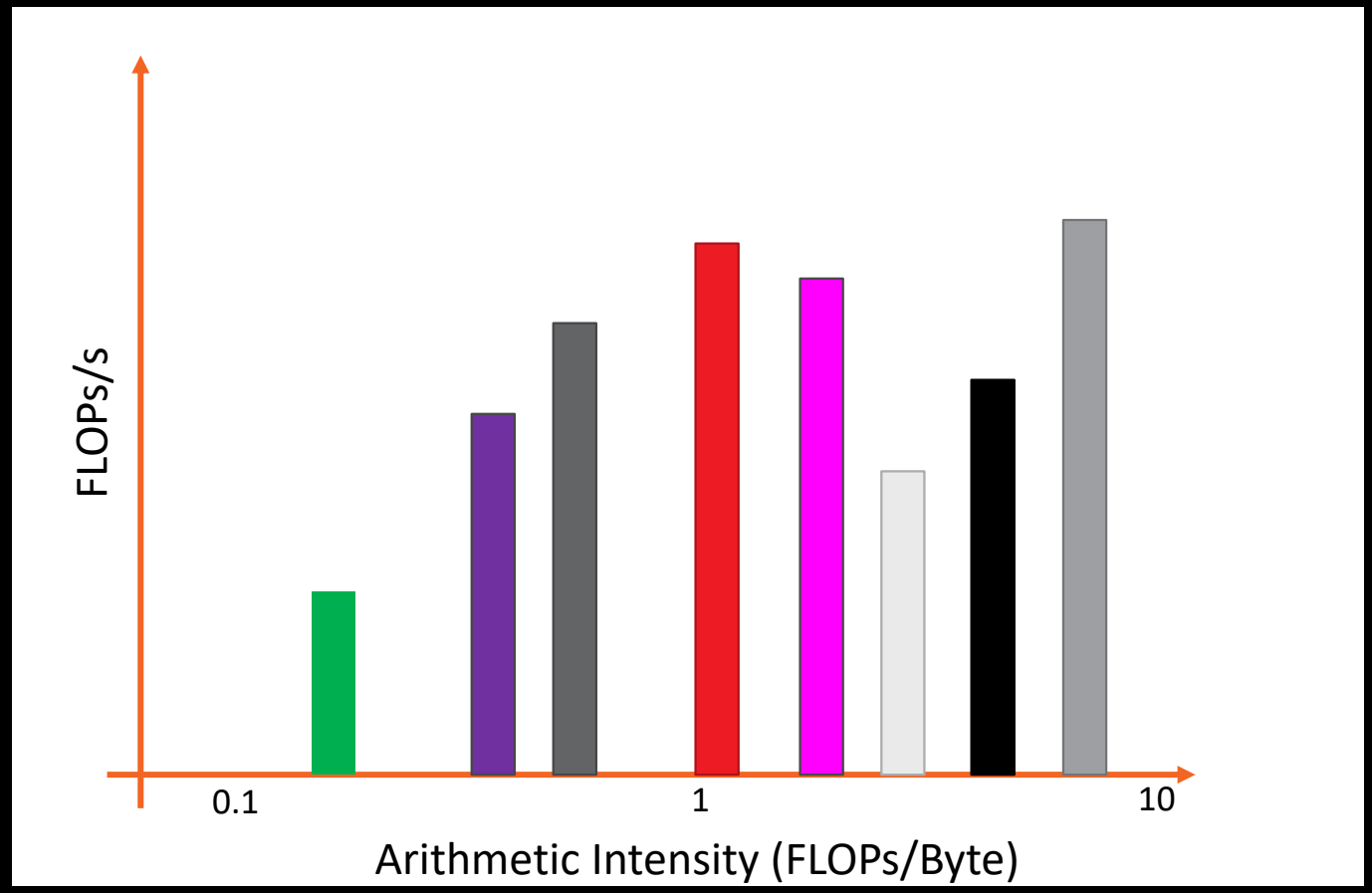
Background – What is “Good” Performance?

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



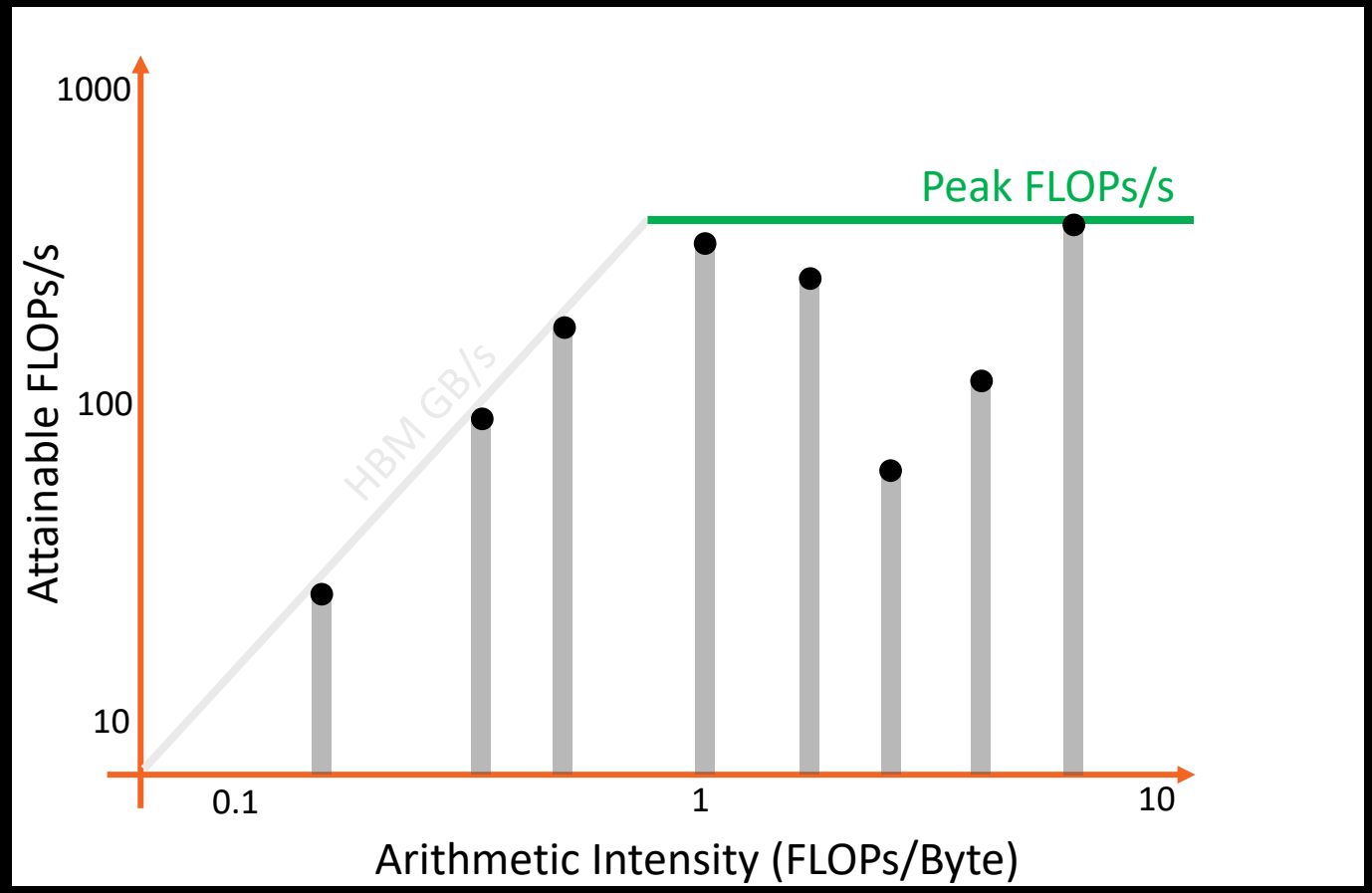
Background – What is “Good” Performance?

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



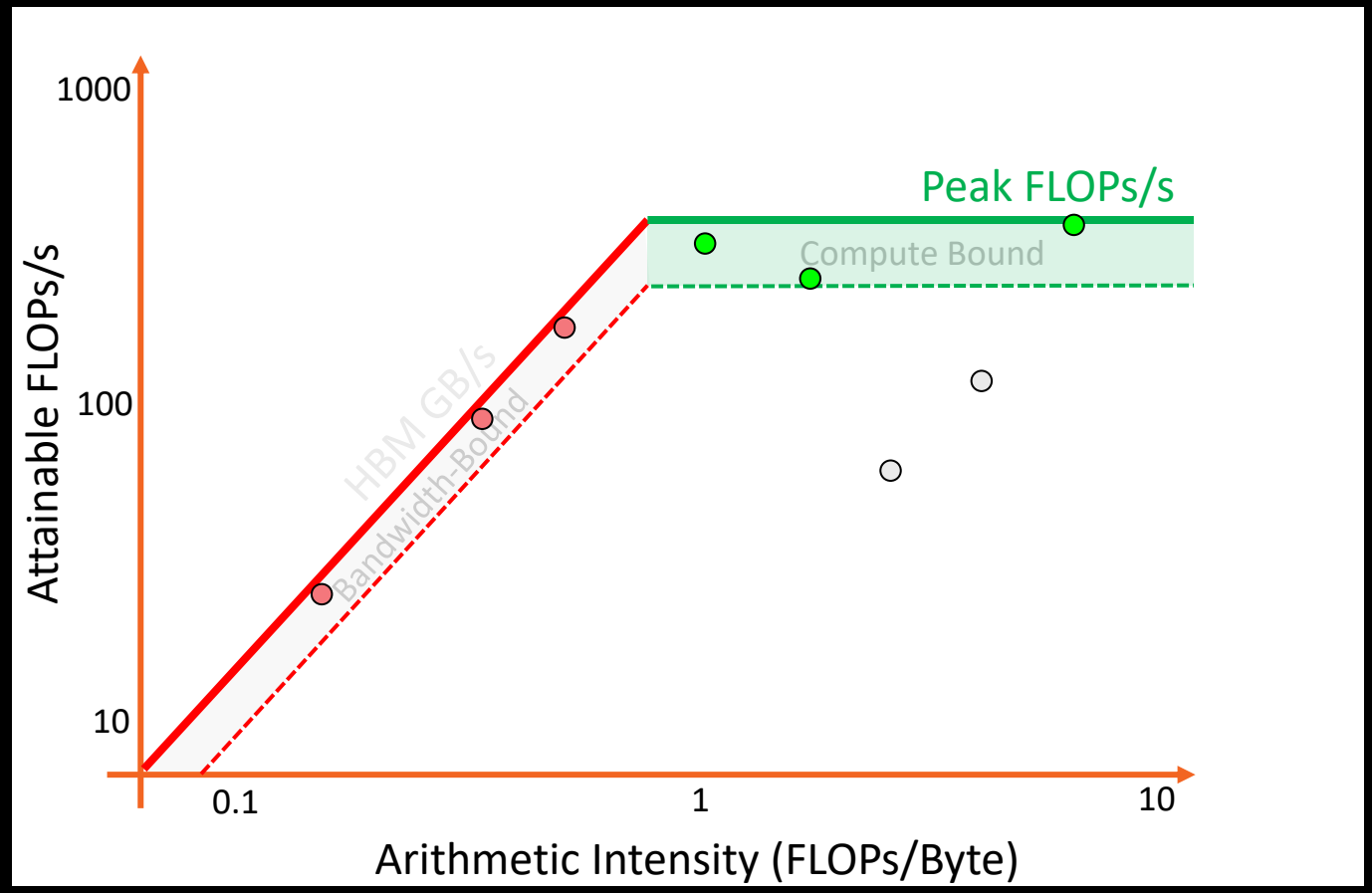
Background – What is “Good” Performance?

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



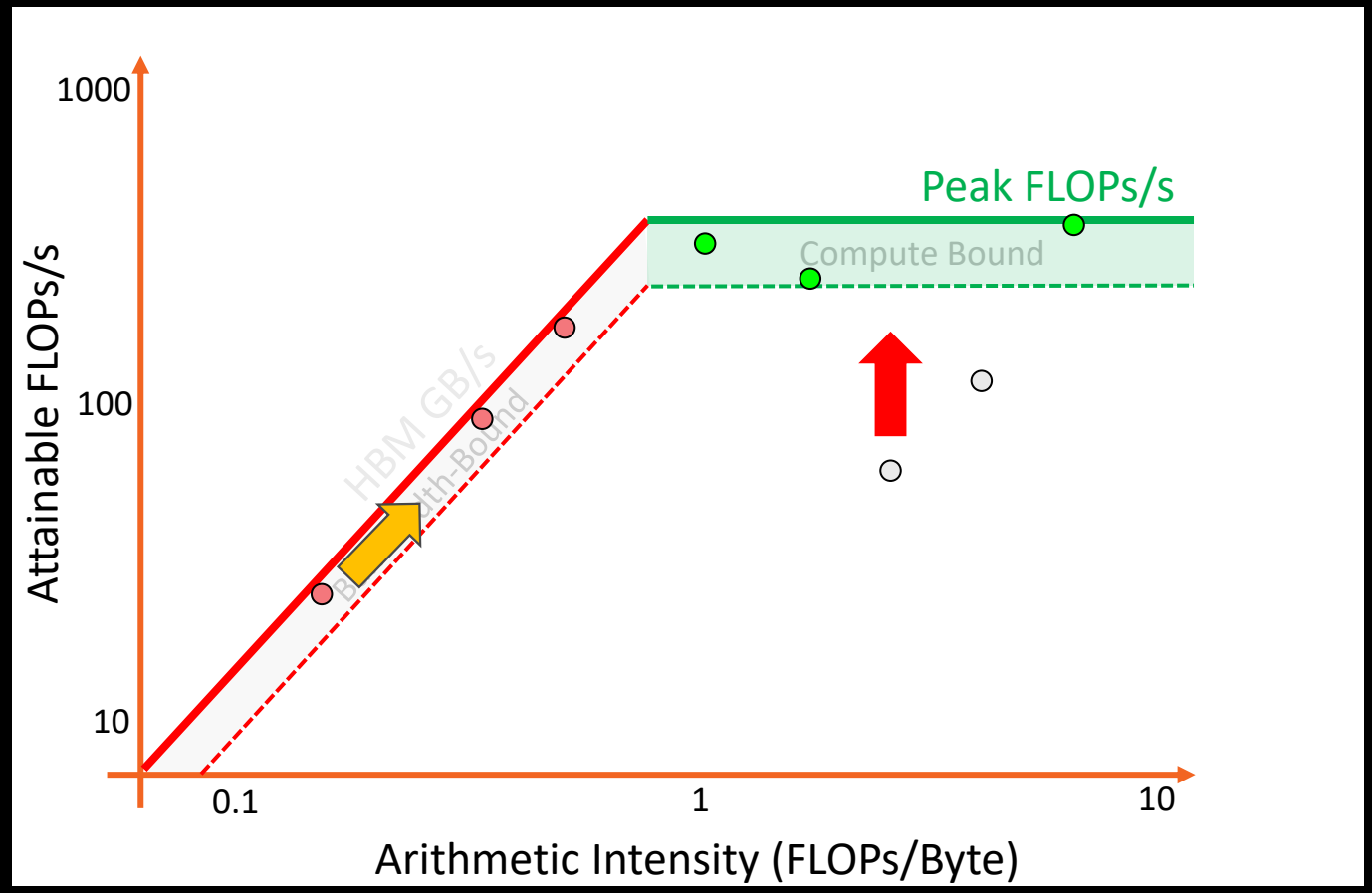
Background – What is “Good” Performance?

- 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



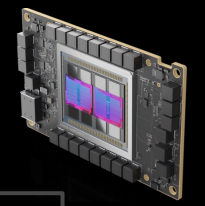
Background – What is “Good” Performance?

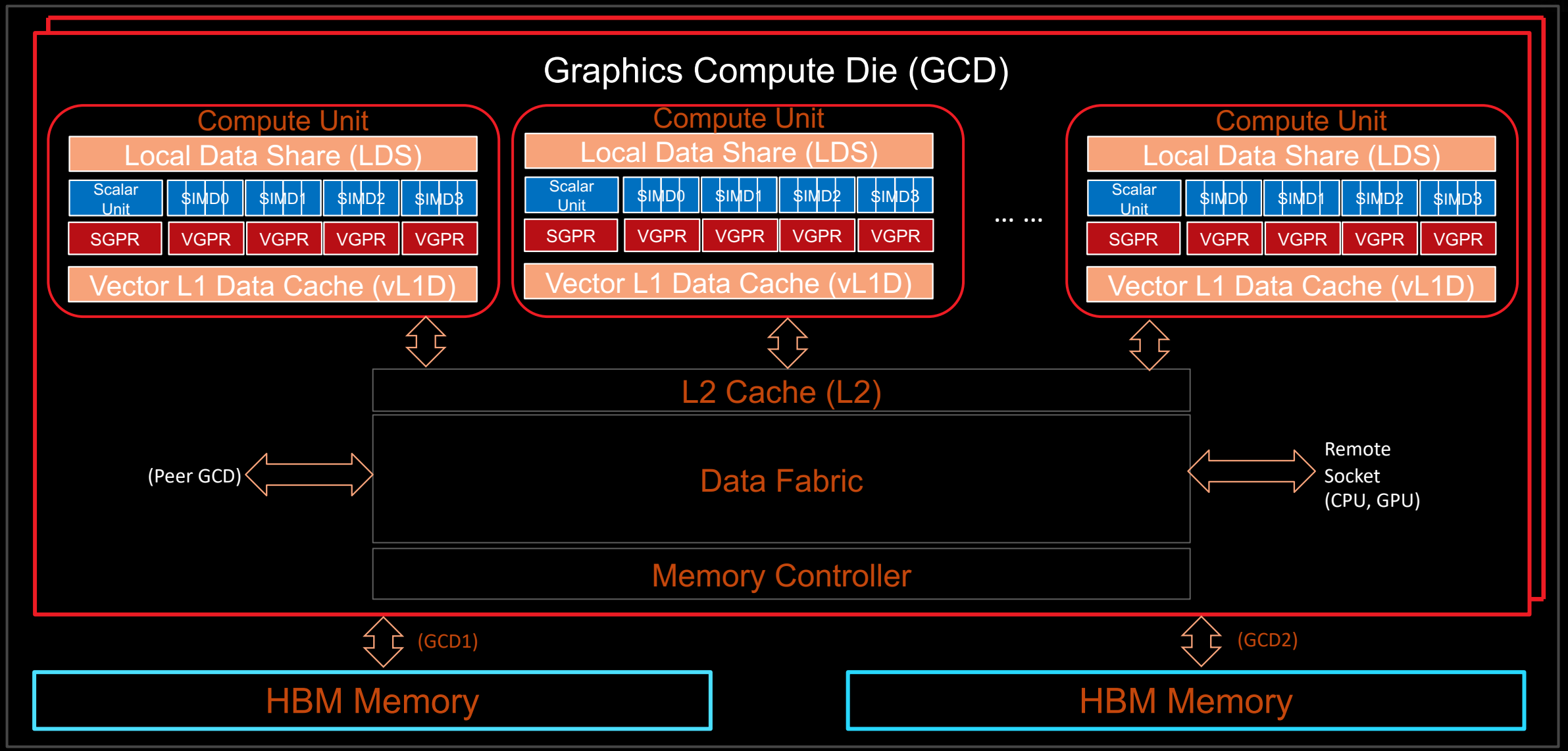
- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities
  - Kernels near the roofline are making good use of computational resources
    - Kernels can have low performance (FLOPS/s), but make good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



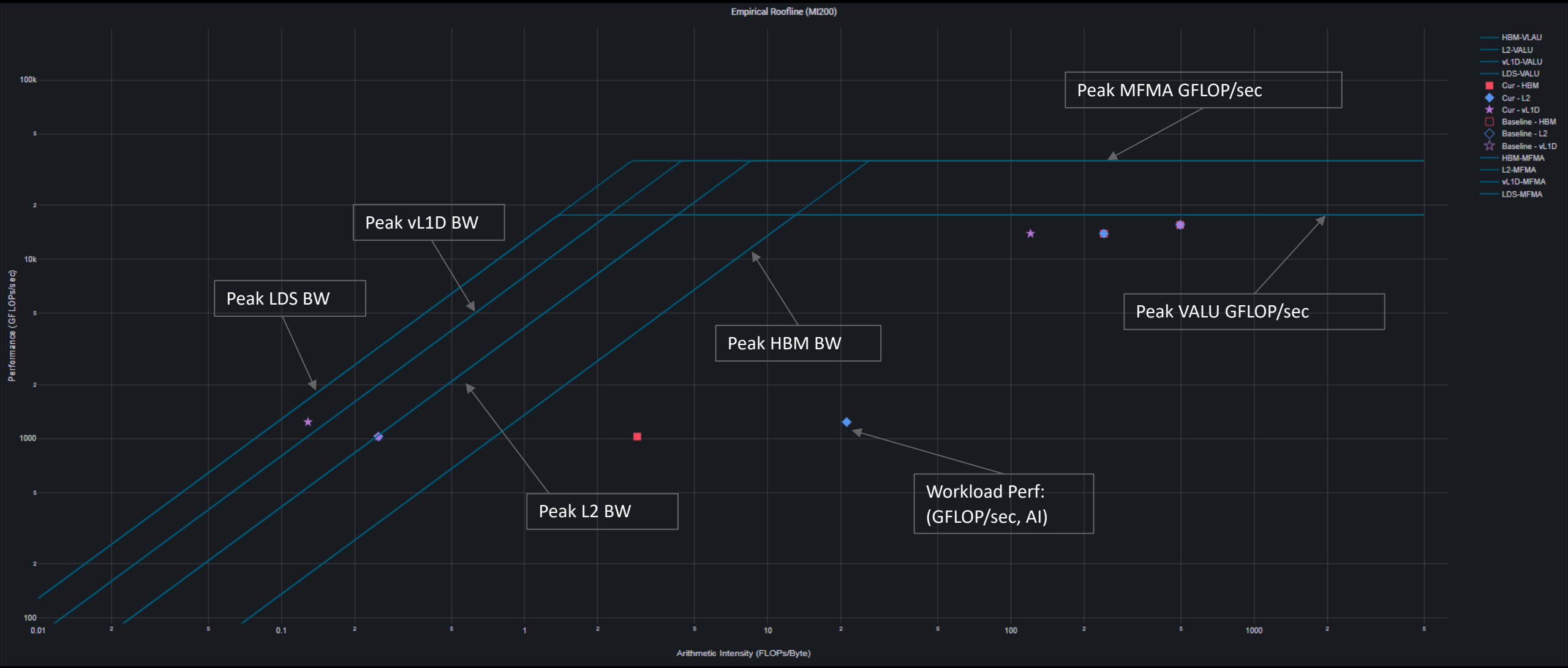
Roofline Calculations on AMD Instinct™ MI200 GPUs

Overview - AMD Instinct™ MI200 Architecture





Empirical Hierarchical Roofline on MI200 - Overview



Guided Exercises

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

Guided Exercises: Logistics/Preamble

- To accommodate the virtual setting and attendees with varied access to Omniperf:
  - I'll read through the slides without waiting for everyone to finish working through each exercise
  - If you have access to a system with Omniperf, clone the repo and start working through the exercises:
    - git clone https://github.com/amd/HPCTrainingExamples/ .
    - The READMEs contain all of what I'm saying and include platform-specific instructions for this training in the top-level directory

- We have used Omniperf 2.0.1 to generate output for these slides:
  - Behavior may differ if using a different version of Omniperf (e.g. 1.0.10)
  - Generally, building stable releases is the best practice

- Some numbers shown in the exercises and these slides were generated using MI210 accelerators

- Implementations in these exercises are **not** fully-optimized kernels

Guided Exercises: Representative Optimization Tasks

- The Exercises are roughly in order of ease of development effort and performance impact:
  - Exercise 1: Verify Reasonable Launch Parameters
  - Exercise 2: Attempt to Cache Data in Shared Memory
  - Exercise 3: Determining a Source of Unexpected Resource Usage
  - Exercise 4: Verifying Efficient Data Access Patterns
  - Exercise 5: Analyzing an Algorithmic Change

- The underlying code is kept simple to emphasize the optimization techniques

- These slides are intended as a “Cheat Sheet” starting point providing:
  - Omniperf commands to filter through output for common optimization concerns
  - Some optimization direction given certain Omniperf output

Guided Exercises: Optimizing a yAx Kernel

- We'll be looking at a relatively simple kernel that solves the same problem in each exercise, yAx
  - yAx is a vector-matrix-vector product that can be implemented in serial as:

```
double result = 0.0;
for (int i = 0; i < n; i++){
  double temp = 0.0;
  for (int j = 0; j < m; j++){
    temp += A[i*m + j] * x[j];
  }
  result += y[i] * temp;
}
```

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

Exercise 1: First Things First, Generate a Roofline

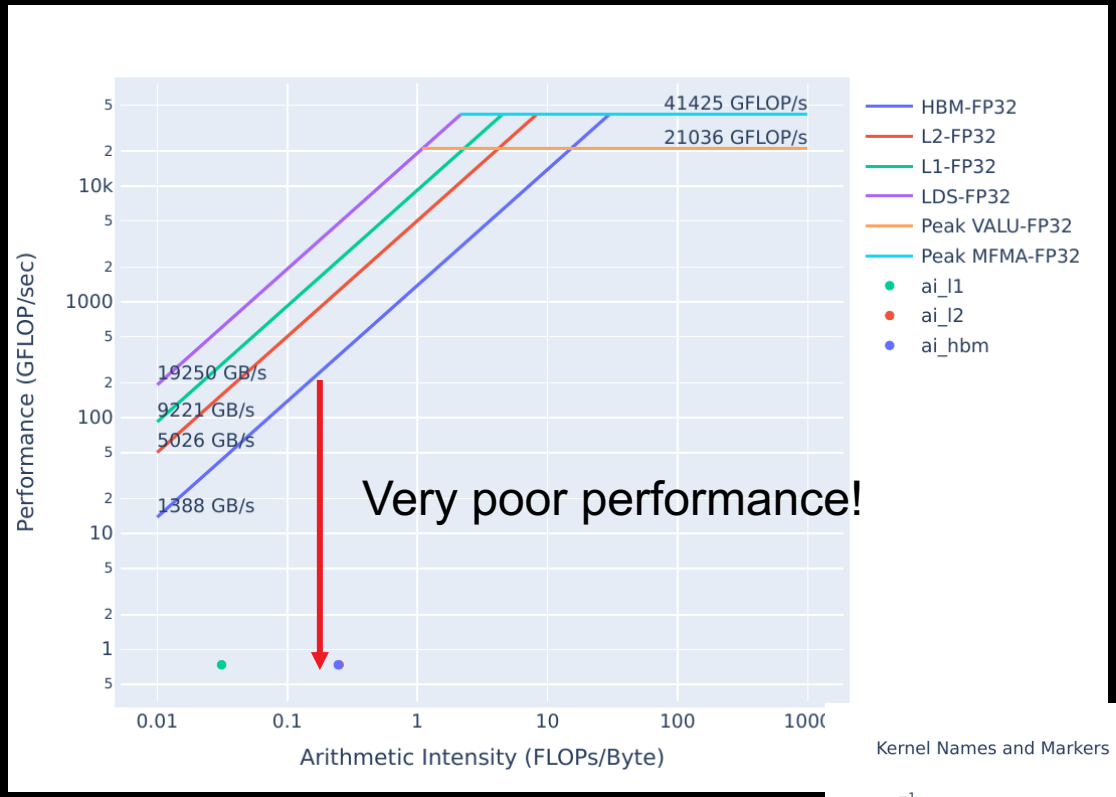
- Run this command to generate roofline plots and a legend for each kernel (in PDF form):
  - `omniperf profile –n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
    - The files will appear in the `./workloads/problem_roof_only/MI200` folder.
    - `--roof-only` generates PDF roofline plots, and does **not** generate any non-roofline profiling data
    - `--kernel-names` generates a PDF showing which kernel names correspond to which icons in the roofline

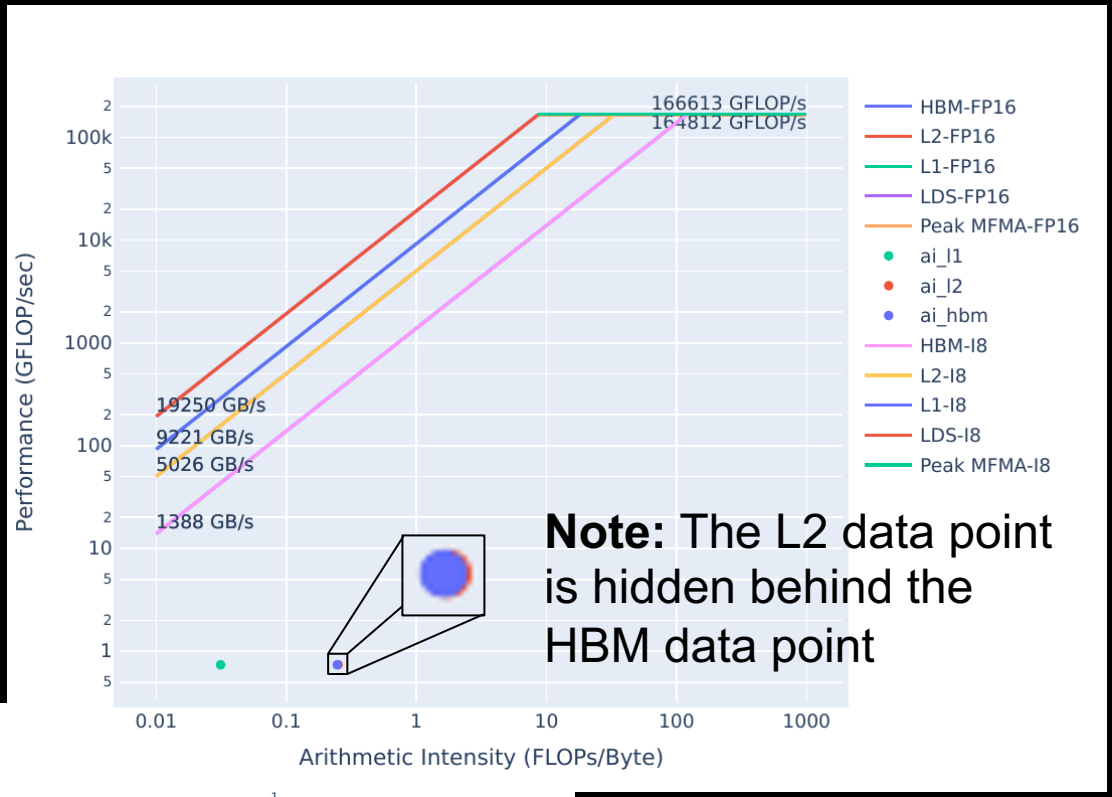
- Rooflines are a useful tool in determining which kernels are good optimization targets
  - They are only one perspective of performance: runtime of the kernel cannot be inferred from the roofline

- Generated PDF roofline plots can have overlapping data points but should still be instructive
  - There are fixes to this, but they may be difficult to setup for different cluster installations
  - Generating the PDF plots from the command line interface should always work

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

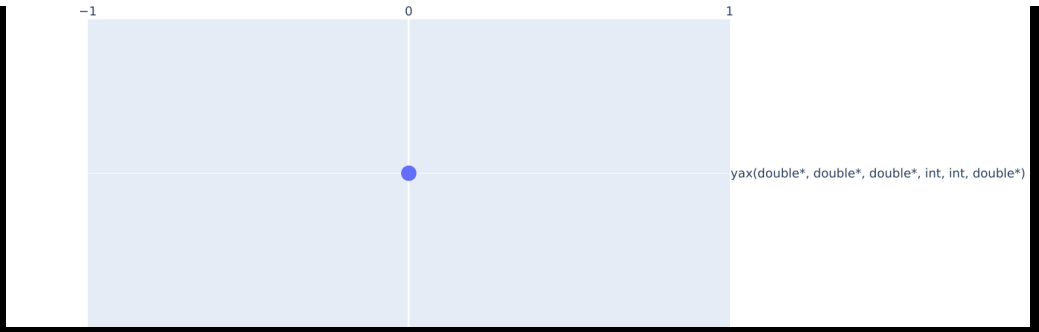
Exercise 1: Problem Roofline Plots





Note: The L2 data point is hidden behind the HBM data point

Kernel legend



Exercise 1: Prep to use Omniperf to Find Kernel Launch Parameters

- Launch parameters are given at the time of the kernel launch, as in lines 49 and 54:
  - `yax<<<grid,block>>>(y,A,x,n,m,result);`
    - Where grid and block are the kernel yax's launch parameters
  - In problem, `grid = (4,1,1)`, and `block = (64,1,1)`
  - In solution, `grid = (2048,1,1)`, and `block = (64,1,1)`

- Sometimes the launch parameters for a given kernel can be obfuscated

- Omniperf can easily show launch parameter information regardless of the code
  - You just need the dispatch ID

- To generate profiling data, use the commands:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`
    - `--no-roof` saves time by not generating roofline data – profile commands can take a while

- **Real benchmarks can take prohibitively long to profile** – use smaller representative problems if possible

Exercise 1: CLI Omniperf Comparisons are Easy

```
omniperf analyze -p workloads/problem/MI200 -p workloads/solution/MI200 --dispatch 1 --block 7.1.0 7.1.1 7.1.2
```

```
--------
Analyze
--------
```

```
--------------------------------------------------------------------------------
0. Top Stat
```

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	754934306.50	69702016.5 (-90.77%)	754934306.50	69702016.5 (-90.77%)	754934306.50	69702016.5 (-90.77%)	100.00	100.0 (0.0%)

10.8x speedup

```
--------------------------------------------------------------------------------
7. Wavefront
7.1 Wavefront Launch Stats
```

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit
7.1.0	Grid Size	256.00	131072.0 (51100.0%)	256.00	131072.0 (51100.0%)	256.00	131072.0 (51100.0%)	Work items
7.1.1	Workgroup Size	64.00	64.0 (0.0%)	64.00	64.0 (0.0%)	64.00	64.0 (0.0%)	Work items
7.1.2	Total Wavefronts	4.00	2048.0 (51100.0%)	4.00	2048.0 (51100.0%)	4.00	2048.0 (51100.0%)	Wavefronts

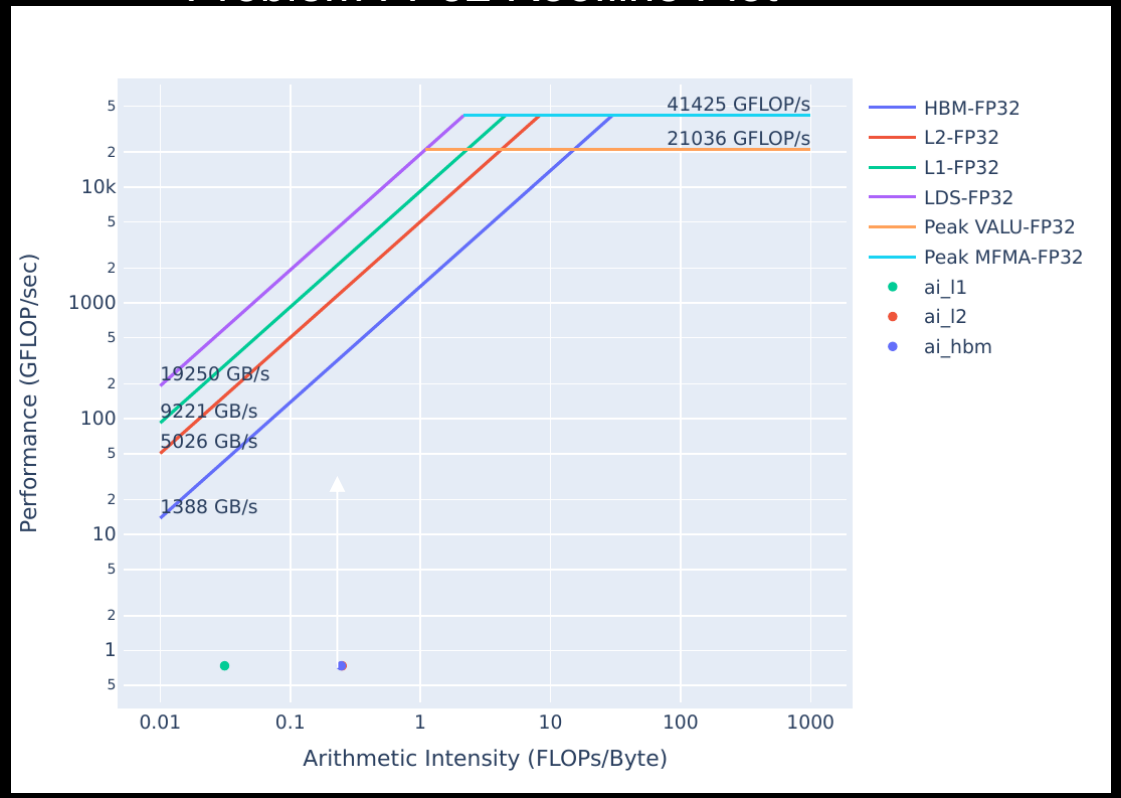
Increased launched wavefronts, which increases Grid Size

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

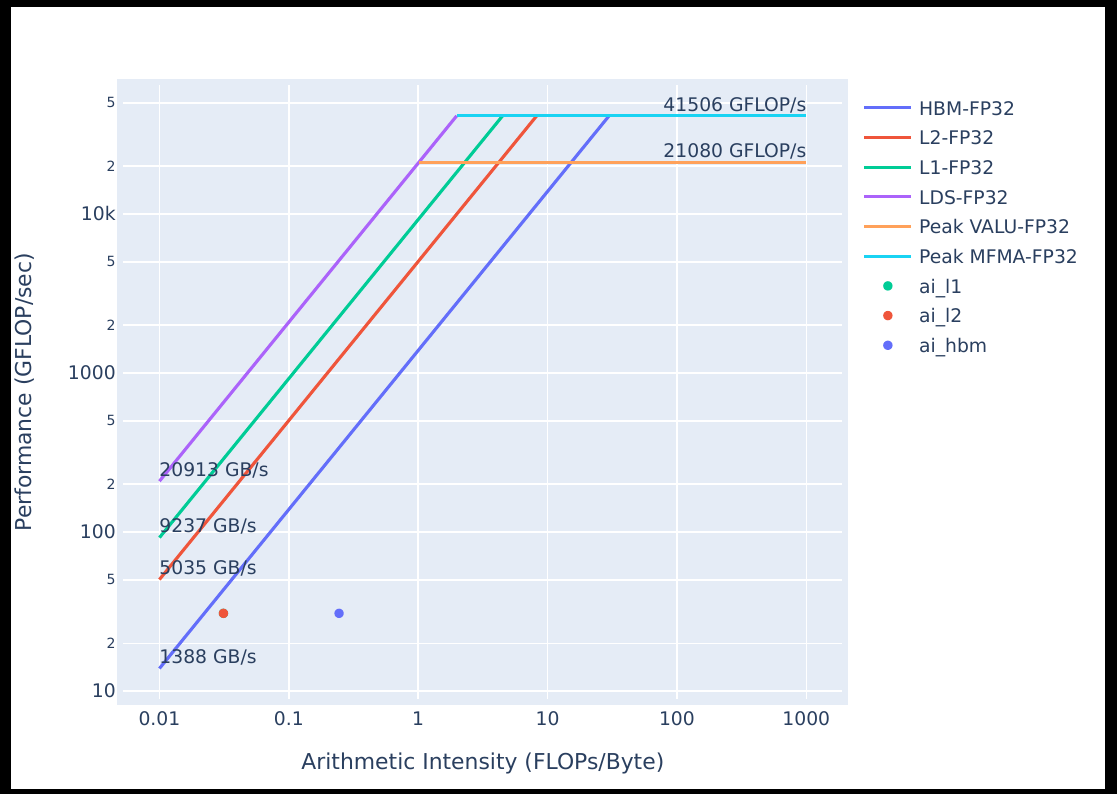
These slides always put `problem` as the baseline, and `solution` as the comparative

Exercise 1: Comparing Problem and Solution Roofline Plots

Problem FP32 Roofline Plot



Solution FP32 Roofline Plot



Generally, moving **up** and to the **right** is good.

Exercise 1: It's Easy to Check Launch Parameters with Omniperf

- Use this omniperf command to check launch parameters:
  - `omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 7.1.0 7.1.1 7.1.2`
    - Shows the launch parameters of the kernel with dispatch ID 1
    - `--block` filters the output to **only** show these launch parameters

- Good launch parameters are essential to a performant GPU kernel
  - Determining which parameters give the best performance usually requires experimenting

- It can be difficult to track down where launch parameters are set in code

- Omniperf can easily show the launch parameters of a kernel
  - Need the dispatch ID or index given by `--list-stats`
  - `--list-stats` index can be passed to `-k` as in:
    - `omniperf analyze -p workloads/problem/MI200 -k 0 -metric 7.1.0 7.1.1 7.1.2`

- **Note:**
  - These metric numbers are for Omniperf 1.0.10

Exercise 2: Diagnosing a Shared Memory Occupancy Limiter

- Using LDS (Local Data Store – Shared Memory) to cache re-used data can be an effective optimization strategy
- Using **too much** LDS can restrict occupancy however, and reduce performance
- Line 12 in problem.cpp shows the allocation of LDS:
  - `__shared__ double tmp[fully_allocate_lds];`
- There are two solutions:
  - `solution-no-lds` removes the LDS allocation, and thus the occupancy limiter
  - `solution` reduces the size of the LDS allocation, removes occupancy limiter, and is faster than `solution-no-lds`
    - This is the solution used to generate the Omniperf output in the next slide
- Omniperf makes it easy to determine if LDS allocations restrict occupancy, as before profile with:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 2: LDS Occupancy Limiter – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/MI200 -p workloads/solution/MI200 --dispatch 1 --block 2.1.15 6.2.7
```

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	166543303.00	38718894.0 (-76.75%)	166543303.00	38718894.0 (-76.75%)	166543303.00	38718894.0 (-76.75%)	100.00	100.0 (0.0%)

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

4.4x speedup

```
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Peak	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	98.45	476.47 (383.96%)	10.74	Wavefronts	3520.00	3520.0 (0.0%)	2.80	13.54 (383.98%)

+ ~14% Occupancy (overall)

```
--------------------------------------------------------------------------------
6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation
```

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.7	Insufficient CU LDS	69.53	0.0 (-99.99%)	-69.53	69.53	0.0 (-99.99%)	69.53	0.0 (-99.99%)	Pct

Sharp decrease in SPI stat

Exercise 2: Use SPI Stats to Determine if LDS Limits Occupancy

- Occupancy limiters can negatively impact performance
- Workgroup manager (SPI) stats in Omniperf indicate whether a kernel resource limits occupancy
- You can get the SPI stat for LDS for a single kernel with:
  - `omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 2.1.15 6.2.7`

Note:

- In current Omniperf release 2.0.1, the SPI “insufficient resource” stats are a percentage of cycles count:
  - If two fields are nonzero, the larger number indicates that resource is limiting occupancy more
- In a coming release, these “insufficient resource” fields are changing to percentages:
  - Large numbers will no longer be expected, but the other points will still hold

Exercise 3: Diagnosing a Register Occupancy Limiter

- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - In this case an `assert` on line 15 causes occupancy to be limited by register usage
  - The solution simply removes the assert

- The types of registers on AMD GPUs are:

  - **VGPRs (Vector General Purpose Registers):** registers that can hold distinct values for each thread in the wavefront

  - **SGPRs (Scalar General Purpose Registers):** uniform across a wavefront. If possible, using these is preferable

  - **AGPRs (Accumulation vector General Purpose Registers):** special-purpose registers for MFMA (Matrix Fused Multiply-Add) operations, or low-cost register spills

- Using too many of one of these register types can impact occupancy and negatively impact performance

- We use the same profile commands to get the profiling data:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 3: Register Occupancy Limiter – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/MI200 -p workloads/solution/MI200 --dispatch 1 --block 2.1.15 6.2.5 7.1.5 7.1.6 7.1.7
```

0. Top Stats
0.1 Top Kernels

	Kernel_Name	Count	Count	Abs Diff	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*) [clone .kd]	1.00	1.0 (0.0%)	0.00	79412646.00	69695296.0 (-12.24%)	79412646.00	69695296.0 (-12.24%)	79412646.00	69695296.0 (-12.24%)	100.00	100.0 (0.0%)

0.2 Dispatch List

	Dispatch_ID	Kernel_Name	GPU_ID
0	1	yax(double*, double*, double*, int, int, double*) [clone .kd]	4

Minor speedup

2. System Speed-of-Light
2.1 Speed-of-Light

Metric_ID	Metric	Avg	Avg	Abs Diff	Unit	Peak	Peak	Pct of Peak	Pct of Peak
2.1.15	Wavefront Occupancy	410.58	416.09 (1.34%)	0.16	Wavefronts	3520.00	3520.0 (0.0%)	11.66	11.82 (1.37%)

Small increase in occupancy

6. Workgroup Manager (SPI)
6.2 Workgroup Manager - Resource Allocation

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
6.2.5	Insufficient SIMD VGPRs	0.04	0.0 (-94.08%)	-0.04	0.04	0.0 (-94.08%)	0.04	0.0 (-94.08%)	Pct

Large decrease in SPI stat

7. Wavefront
7.1 Wavefront Launch Stats

Metric_ID	Metric	Avg	Avg	Abs Diff	Min	Min	Max	Max	Unit
7.1.5	VGPRs	92.00	32.0 (-65.22%)	-60.00	92.00	32.0 (-65.22%)	92.00	32.0 (-65.22%)	Registers
7.1.6	AGPRs	132.00	0.0 (-100.0%)	-132.00	132.00	0.0 (-100.0%)	132.00	0.0 (-100.0%)	Registers
7.1.7	SGPRs	64.00	112.0 (75.0%)	48.00	64.00	112.0 (75.0%)	64.00	112.0 (75.0%)	Registers

Able to use:
Fewer VGPRs,
No AGPRS,
more SGPRs

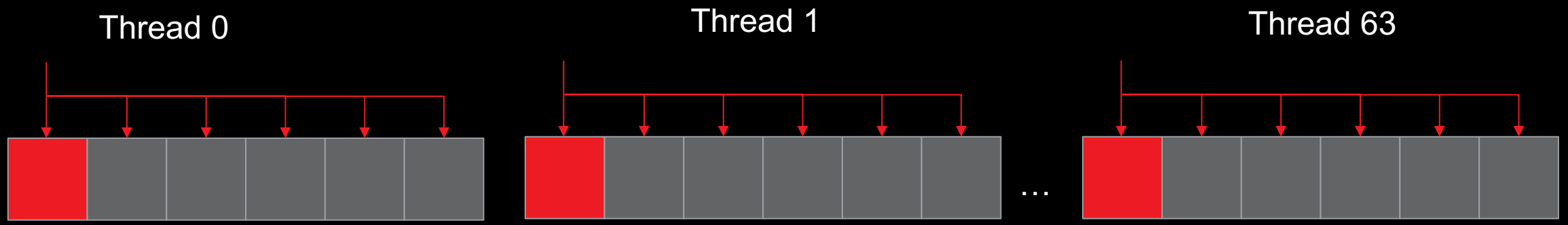
Exercise 3: Register Occupancy Limiter - Takeaways

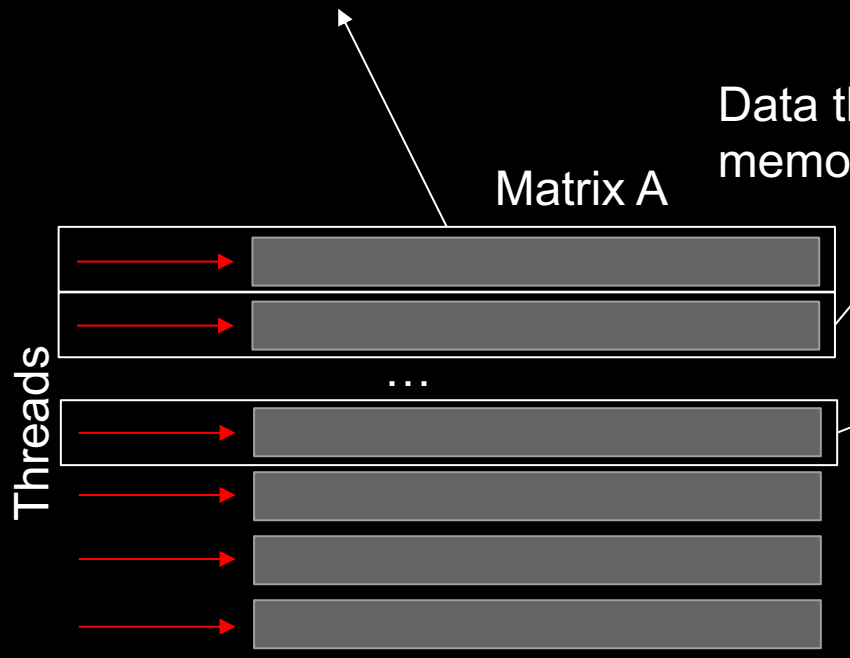
- Seemingly innocuous function calls inside kernels can lead to unexpected performance characteristics
  - Asserts, and even excessive use of math functions in kernels can degrade performance
- In this case the occupancy limit was very minor, despite a large number in the SPI stat
- AGPR usage in the absence of MFMA (Matrix Fused Multiply Add) instructions can indicate degraded performance.
  - Spilling registers to AGPRs, due to running out of VGPRs
- To determine if any SPI “insufficient resource” stats are nonzero, you can do:
  - `omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 6.2`
    - **Note:** This will report more than just all “insufficient resource” fields

Exercise 4: Data Access Patterns are Important to Performance

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

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

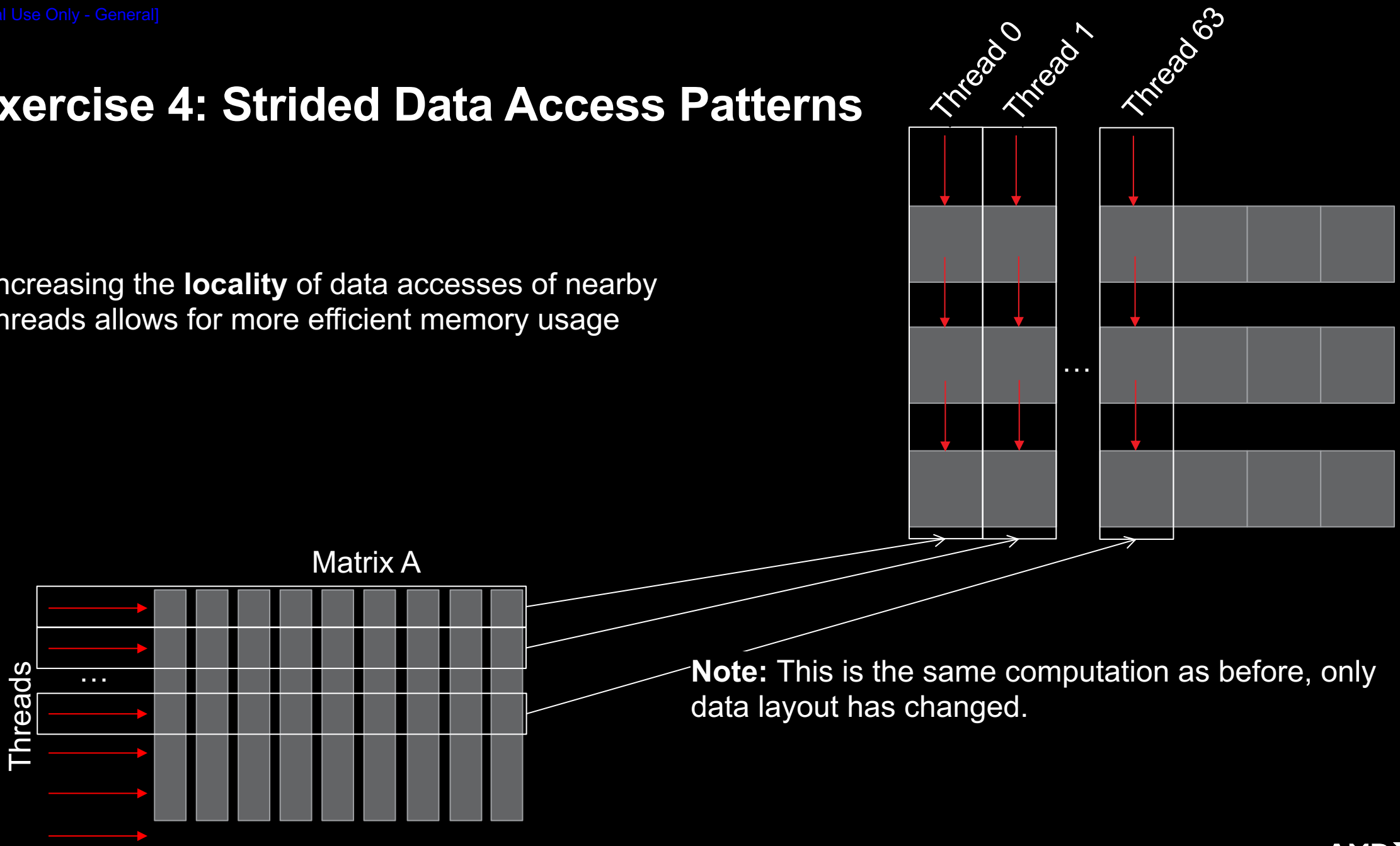




Data that each thread accesses at each step requires striding through memory, which leads to sub-optimal memory system usage.

Exercise 4: Strided Data Access Patterns

Increasing the **locality** of data accesses of nearby threads allows for more efficient memory usage



Note: This is the same computation as before, only data layout has changed.

Exercise 4: Using Omniperf to Diagnose a Strided Data Access Pattern

- This exercise's setup makes it very easy to change the data access pattern
  - Generally, these optimizations can have nontrivial development overhead
  - Re-conceptualizing the data structure can be difficult

- All the solution does is re-work the indexing scheme to better use caches
  - No required change to underlying data, because all the values in y, A, and x are set to 1

- To get started run:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 4: Strided Data Access Pattern – Relevant Omniperf Output

```
omniperf analyze -p workloads/problem/MI200 -p workloads/solution/MI200 --dispatch 1 --block 16.1 17.1
```

```
--------------------------------------------------------------------------------
0. Top Stat
```

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	69875592.00	12469690.5 (-82.15%)	69875592.00	12469690.5 (-82.15%)	69875592.00	12469690.5 (-82.15%)	100.00	100.0 (0.0%)

5.6x speedup

```
--------------------------------------------------------------------------------
16. Vector L1 Data Cache
16.1 Speed-of-Light
```

Index	Metric	Value	Value	Unit
16.1.0	Buffer Coalescing	25.00	25.0 (0.0%)	Pct of peak
16.1.1	Cache Util	87.80	98.08 (11.7%)	Pct of peak
16.1.2	Cache BW	8.69	12.18 (40.19%)	Pct of peak
16.1.3	Cache Hit	0.00	49.98 (inf%)	Pct of peak

+ ~50% in L1 hit

```
--------------------------------------------------------------------------------
17. L2 Cache
17.1 Speed-of-Light
```

Index	Metric	Value	Value	Unit
17.1.0	L2 Util	98.74	98.39 (-0.36%)	Pct
17.1.1	Cache Hit	93.45	0.52 (-99.44%)	Pct
17.1.2	L2-EA Rd BW	125.69	688.98 (448.16%)	Gb/s
17.1.3	L2-EA Wr BW	0.00	0.0 (inf%)	Gb/s

L2 Cache Hit decreases sharply, Read BW from HBM increases by ~5x

The solution better uses the L1, but our L2 hit rate has degraded, which points to a deficiency in our algorithm

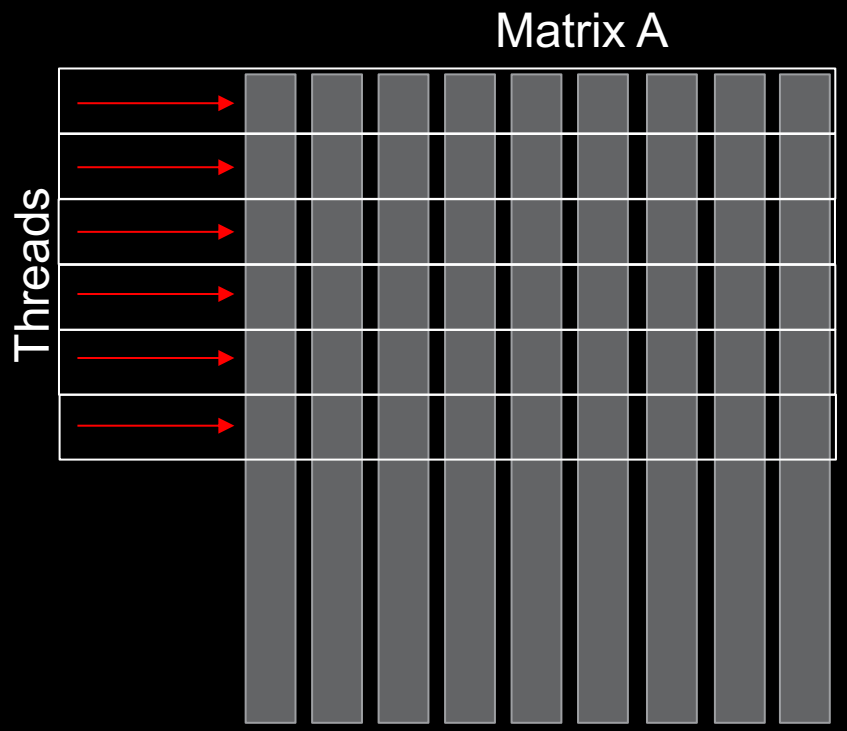
Exercise 4: Omniperf Speed-of-Light Cache Access Statistics

- This Omniperf command will show high-level details about L1 and L2 cache accesses:
  - `omniperf analyze -p workloads/problem/MI200 --dispatch 1 --block 16.1 17.1`
- Ensuring better data locality will generally provide better performance
- In this case, we start hitting in the L1 cache, rather than having to go out to L2 for everything
- **Note:** In a real code, optimizations of this type likely have much more development overhead
  - Need to change how the data structure is indexed everywhere

Exercise 5: Algorithmic Optimizations

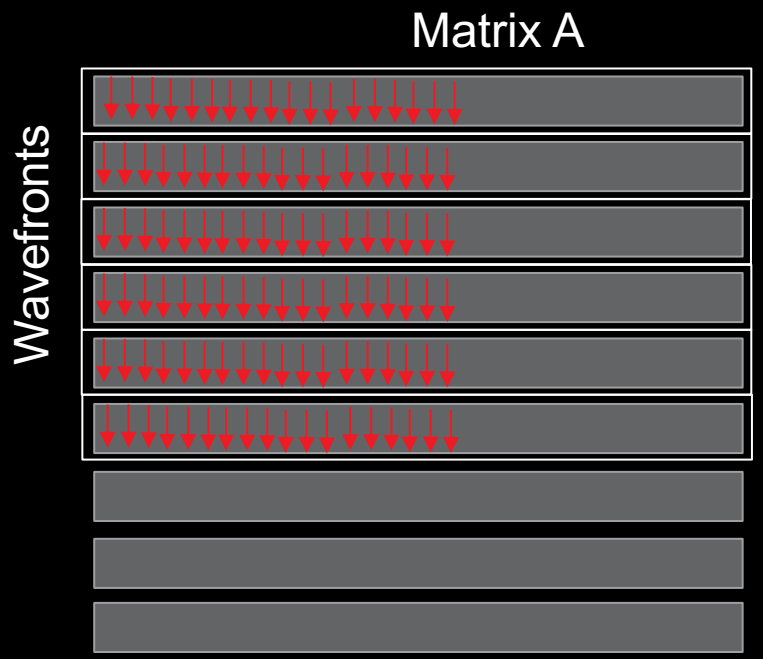
- These types of optimizations are the most difficult to execute
  - Generally, it is difficult to determine if the runtime of one algorithm will be faster than another
- We start with the solution from last exercise as our problem
  - Speed-of-light cache statistics showed that we had ~0% hit rate in the L2, could it be better?
- Our initial algorithm is naïve in terms of parallelization:
  - Each thread computes the sum of a row
- Exposing more parallelism is possible and should get us more performance in this case

Exercise 5: Algorithmic Optimizations



In our current algorithm, each thread computes the sum of a single row

Exercise 5: Algorithmic Optimizations



In a more efficient implementation, wavefronts have multiple threads sum up the rows in parallel, using shared memory to reduce partial sums

Note: The original data layout allows the wavefronts to avoid striding memory

Exercise 5: Using Omniperf to Evaluate an Algorithmic Optimization

- The strided data access pattern issue is everywhere
  - This solution gets about 2x faster when the data layout is switched to optimize locality

- Though the solution shows a **29x speedup** from the problem, cache speed-of-light stats aren't convincing
  - The rooflines for these problems do not tell the full performance story either

- Running the solution shows it is much faster, but does it use the caches more efficiently?

- To get started, run:
  - `omniperf profile –n problem --no-roof -- ./problem.exe`
  - `omniperf profile –n solution --no-roof -- ./solution.exe`

Exercise 5: Sometimes the Full Story is in the Details

```
omniperf analyze -p workloads/problem/MI200 -p workloads/solution/MI200 --dispatch 1 --block 16.3 17.2 17.3
```

0. Top Stat

	KernelName	Count	Count	Sum(ns)	Sum(ns)	Mean(ns)	Mean(ns)	Median(ns)	Median(ns)	Pct	Pct
0	yax(double*, double*, double*, int, int, double*)	1.00	1.0 (0.0%)	12443928.00	408316.0 (-96.72%)	12443928.00	408316.0 (-96.72%)	12443928.00	408316.0 (-96.72%)	100.00	100.0 (0.0%)

~29x faster

16. Vector L1 Data Cache

16.3 L1D Cache Accesses

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
16.3.0	Total Req	524368.00	16448.0 (-96.86%)	524368.00	16448.0 (-96.86%)	524368.00	16448.0 (-96.86%)	Req per wave	- ~32x
...									
16.3.5	Cache Accesses	131140.00	4097.0 (-96.88%)	131140.00	4097.0 (-96.88%)	131140.00	4097.0 (-96.88%)	Req per wave	- ~32x
16.3.6	Cache Hits	65538.00	2864.0 (-95.63%)	65538.00	2864.0 (-95.63%)	65538.00	2864.0 (-95.63%)	Req per wave	
16.3.7	Cache Hit Rate	49.98	69.9 (39.87%)	49.98	69.9 (39.87%)	49.98	69.9 (39.87%)	Pct	**+ ~40%**
...									

17. L2 Cache

17.2 L2 - Fabric Transactions

Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
17.2.0	Read BW	4194916.56	65688.69 (-98.43%)	4194916.56	65688.69 (-98.43%)	4194916.56	65688.69 (-98.43%)	Bytes per wave	- ~64x
...									

17.3 L2 Cache Accesses

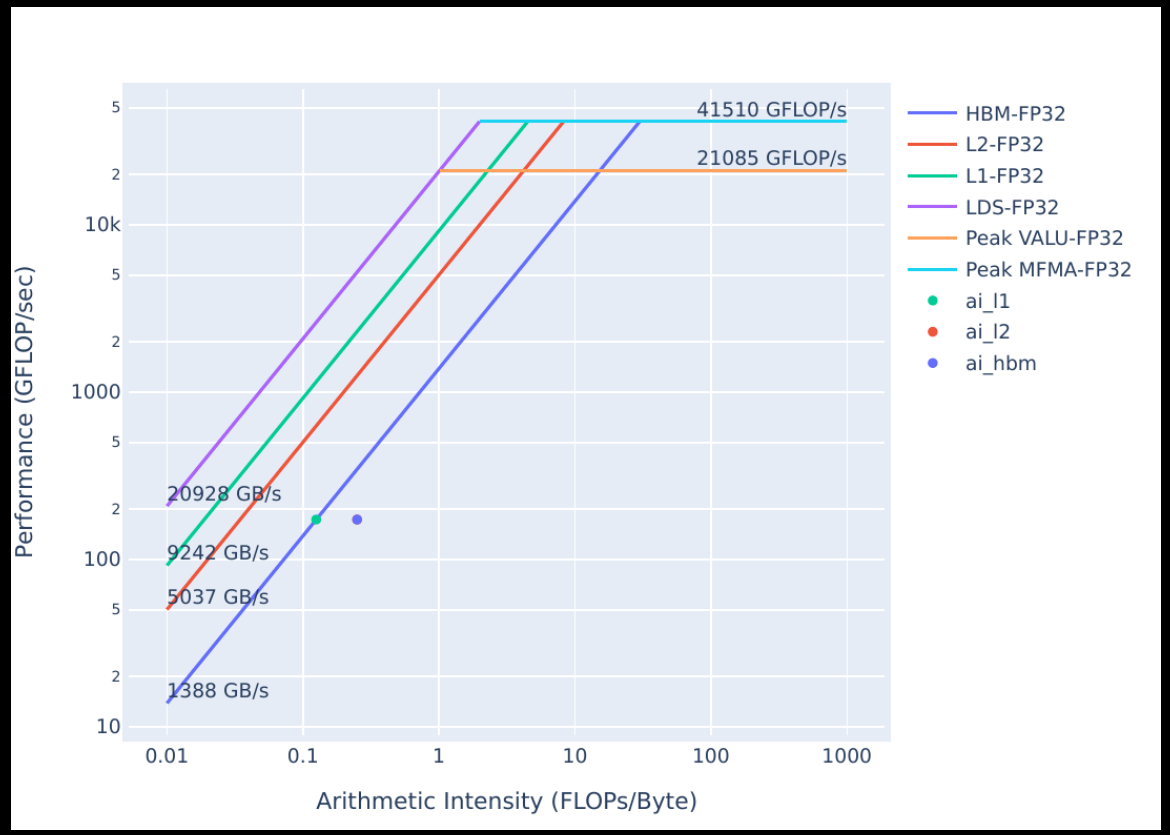
Index	Metric	Avg	Avg	Min	Min	Max	Max	Unit	
17.3.0	Req	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	32945.33	617.41 (-98.13%)	Req per wave	- ~53x
...									
17.3.6	Hits	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	171.28	104.03 (-39.27%)	Hits per wave	
17.3.7	Misses	32774.06	513.38 (-98.43%)	32774.06	513.38 (-98.43%)	32774.06	513.38 (-98.43%)	Misses per wave	- ~64x
17.3.8	Cache Hit	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	0.52	16.85 (3140.15%)	Pct	**Large relative gain, + ~16% overall**
...									

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

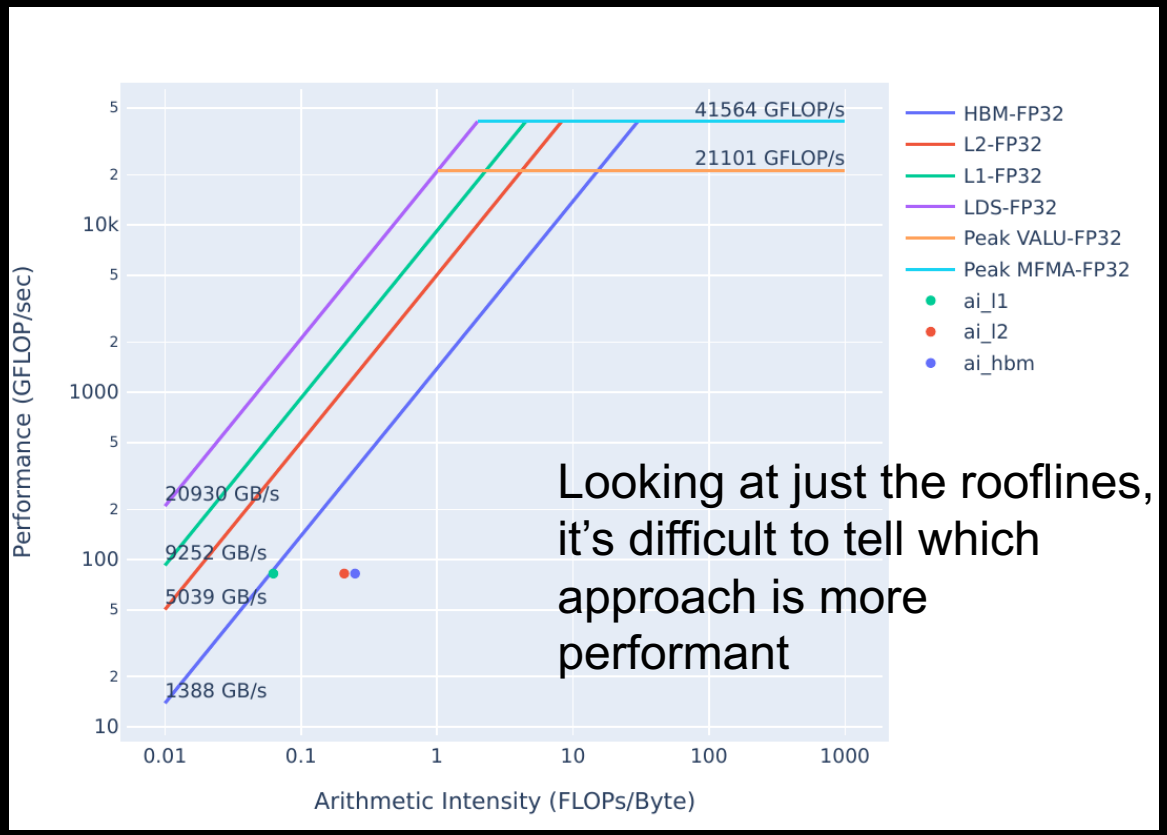
Exercise 5: It Can Be Hard to Compare Rooflines Between Algorithms

- `omniperf profile -n problem_roof_only --roof-only --kernel-names -- ./problem.exe`
- `omniperf profile -n solution_roof_only --roof-only --kernel-names -- ./solution.exe`

Problem FP32 Roofline



Solution FP32 Roofline



Looking at just the rooflines, it's difficult to tell which approach is more performant

`problem` is closer to being HBM bandwidth bound: It needs to request much more data from HBM than the optimized version

Exercise 5: Omniperf Detailed Cache Statistics - Takeaways

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

- Algorithmic optimizations can be powerful, but are usually time-intensive to design and implement

- It can be difficult to understand the performance differences between algorithms
  - Rooflines can be misleading
  - Assuming correctness is verified, timings don't lie
  - Detailed profiling data can help shed light on the *why* of performance differences

Summary

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

Questions?

ssh <you user>@lumi.csc.fi

https://hackmd.io/@sfantao/lumi-training-oslo2024-basic-examples

https://hackmd.io/@sfantao/lumi-training-oslo2024-advanced-omniperf1
https://hackmd.io/@sfantao/lumi-training-oslo2024-advanced-omniperf2

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.

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