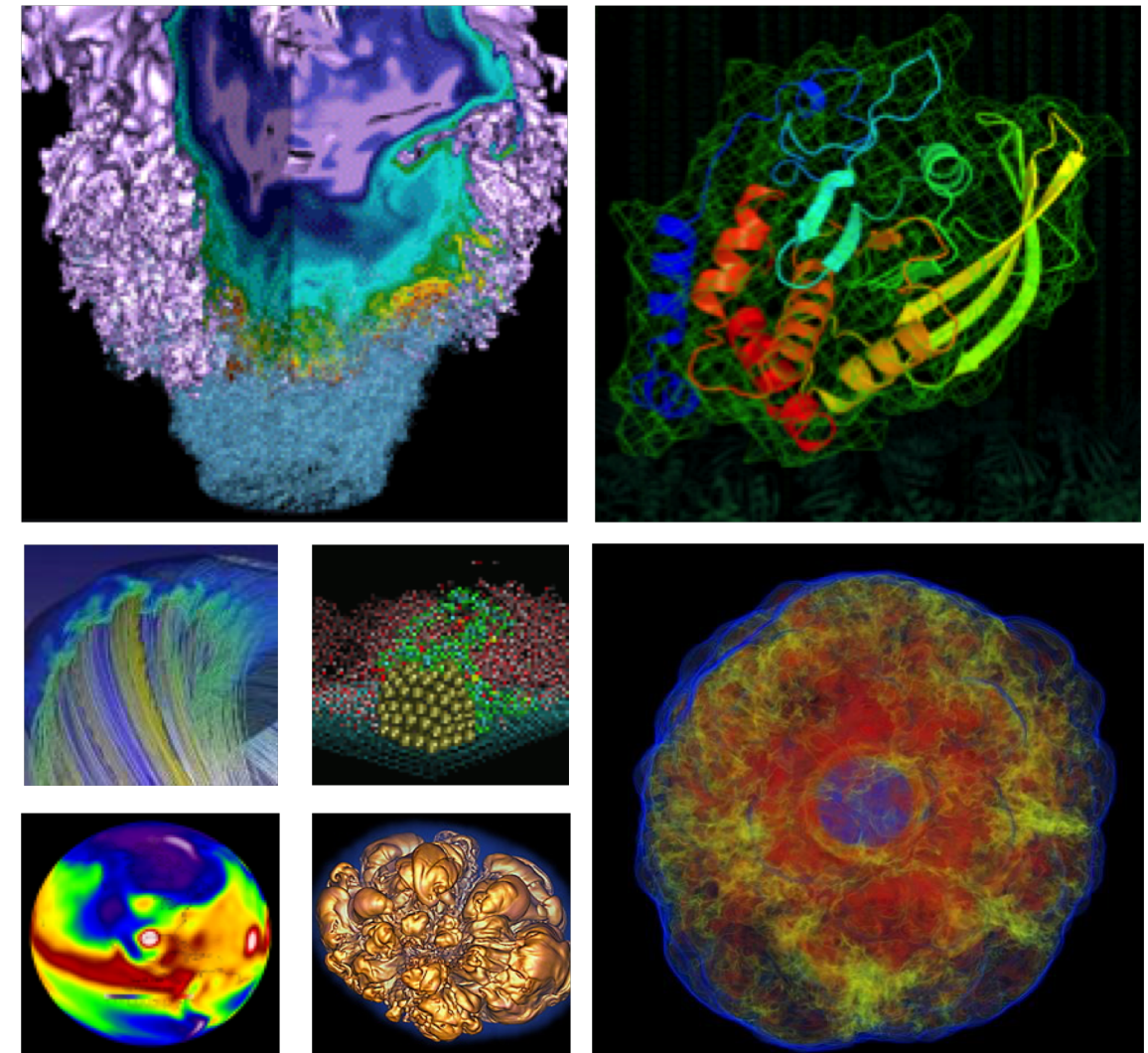


S9624:

Performance Analysis of GPU-Accelerated Applications using the Roofline Model

GTC 2019, San Jose



Charlene Yang
Application Performance Specialist
NERSC, LBNL
cjyang@lbl.gov

Samuel Williams
Senior Staff Scientist
CRD, LBNL
swwilliams@lbl.gov

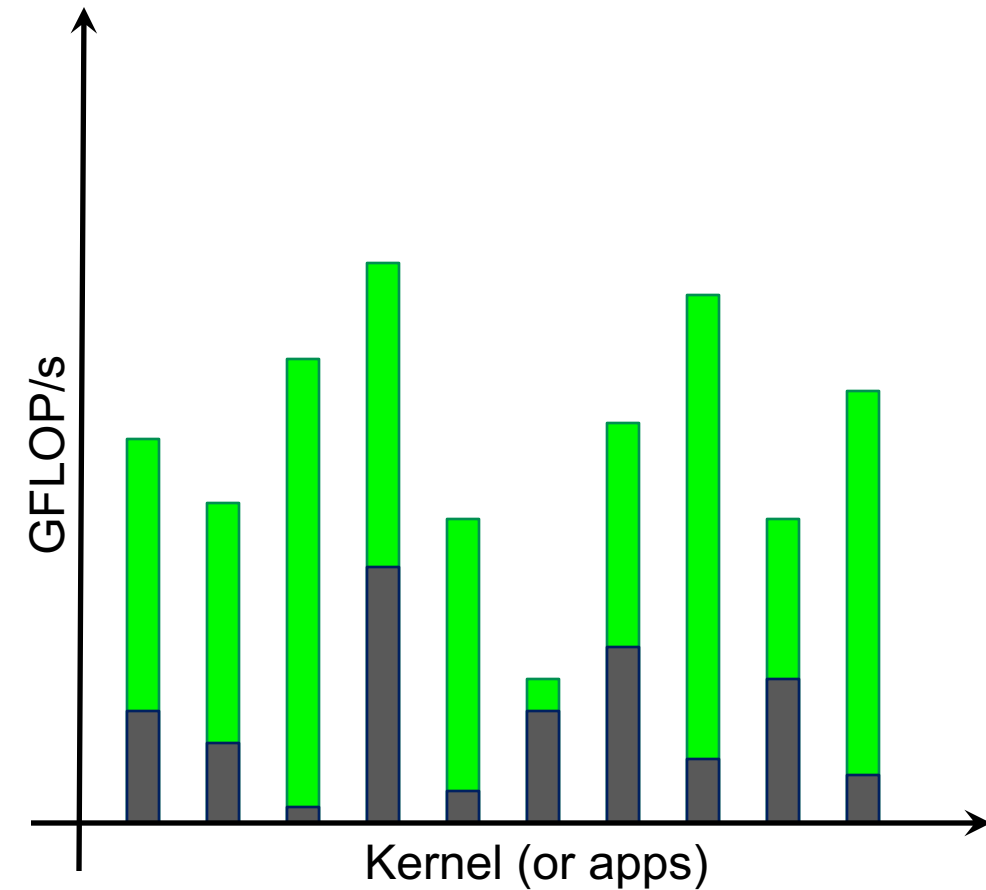
**You just bought a \$10,000
throughput-optimized GPU!**

**Are you making good use of
your investment?**

You could just run benchmarks



- Imagine a mix of benchmarks or kernels...
- GFLOP/s alone may not be particularly insightful
- Moreover, speedup relative to a Xeon may seem random



Making good use of your GPU?



1. Are you operating it in the throughput-limited regime?

- Not sensitive to Amdahl effects
- Not sensitive to D2H/H2D transfers
- Not sensitive to launch overheads
- Not sensitive to latencies

2. If in the throughput-limited regime, are you making good use of the GPU's **compute** and **bandwidth** capabilities?

The Roofline Model



- **Roofline Model** is a throughput-oriented performance model
- Premised on the interplay between FLOP/s, bandwidth, and reuse
- Tracks rates not times
- Independent of ISA and architecture (applies to CPUs, GPUs, Google TPUs, etc...)

Roofline Performance Model

Roofline is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multicore, manycore, or accelerator processor architectures. Rather than simply using percent-of-peak estimates, the model can be used to assess the quality of attained performance by combining locality, bandwidth, and different parallelization paradigms into a single performance figure. One can examine the resultant Roofline figure in order to determine both the implementation and inherent performance limitations.

Arithmetic Intensity

The core parameter behind the Roofline model is Arithmetic Intensity. Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). A BLAS-1 vector-vector increment ($x[i]+y[i]$) would have a very low arithmetic intensity of 0.0417 (N FLOPS / $24N$ Bytes) and would be independent of the vector size. Conversely, FFTs perform $5 \cdot N \cdot \log N$ flops for a N -point double complex transform. If out of place on a write allocate cache architecture, the transform would move at least $48N$ bytes. As such, FFT's would have an arithmetic intensity of $0.104 \cdot \log N$ and would grow slowly with data size. Unfortunately, cache capacities would limit FFT arithmetic intensity to perhaps 2 flops per byte. Finally, BLAS3 and N-Body Particle-Particle methods would have arithmetic intensity grow very quickly.

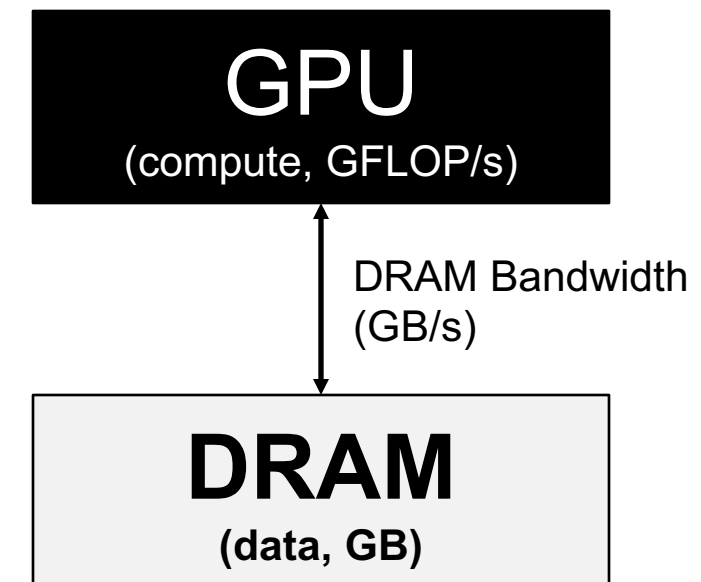
<https://crd.lbl.gov/departments/computer-science/PAR/research/roofline>

Jouppi et al, "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA, 2017.

(DRAM) Roofline



- One could hope to always attain peak performance (GFLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
 - Idealized processor/caches
 - Cold start (data in DRAM)

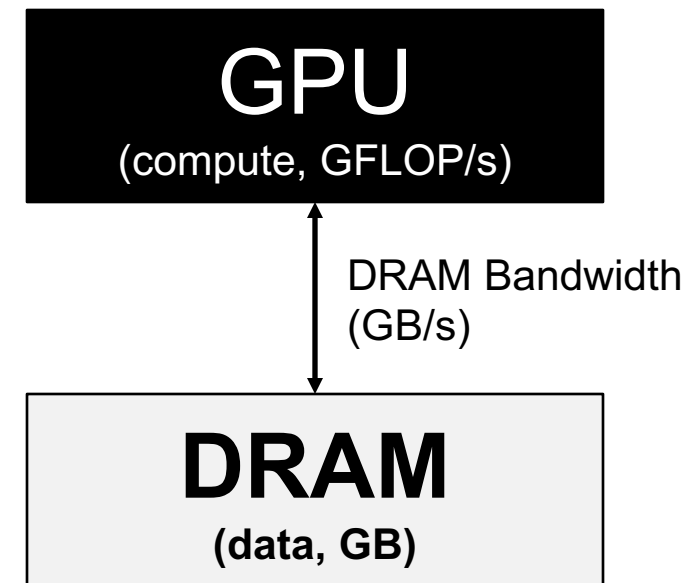


$$\text{Time} = \max \left\{ \begin{array}{l} \# \text{FLOPs} / \text{Peak GFLOP/s} \\ \# \text{Bytes} / \text{Peak GB/s} \end{array} \right.$$

(DRAM) Roofline



- One could hope to always attain peak performance (GFLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
 - Idealized processor/caches
 - Cold start (data in DRAM)



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

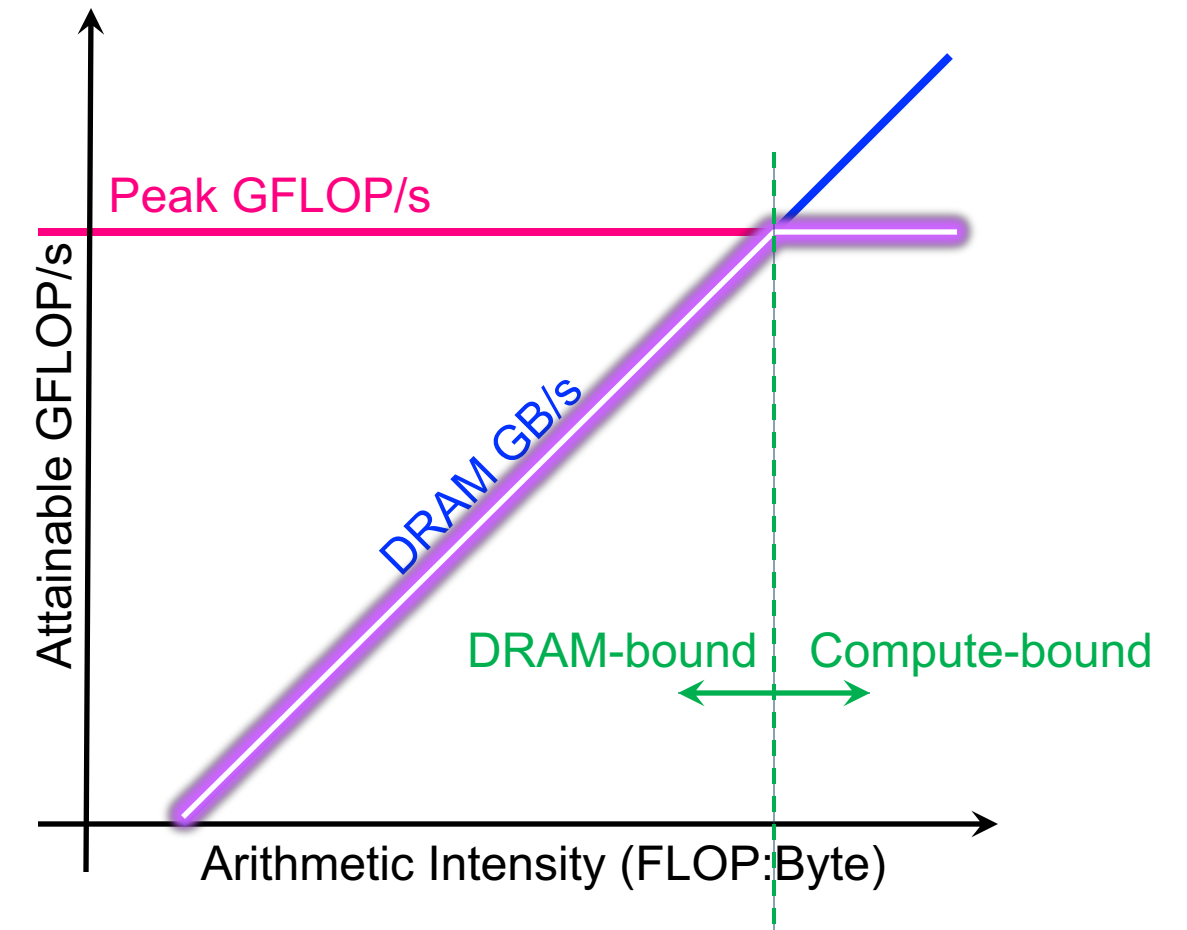
Note, Arithmetic Intensity (AI) = FLOPs / Bytes (as presented to DRAM)

- Arithmetic Intensity is the most important concept in Roofline.
- Measure of data locality (data reuse)
- Ratio of Total FLOPs performed to Total Bytes moved
- For the DRAM Roofline...
 - Total Bytes to/from DRAM and includes all cache and prefetcher effects
 - Can be **very different from total loads/stores** (bytes requested) due to cache reuse

(DRAM) Roofline



- Plot Roofline bound using Arithmetic Intensity as the x-axis
- **Log-log scale** makes it easy to doodle, extrapolate performance along Moore's Law, etc...
- Kernels with AI less than machine balance are ultimately DRAM bound (we'll refine this later...)

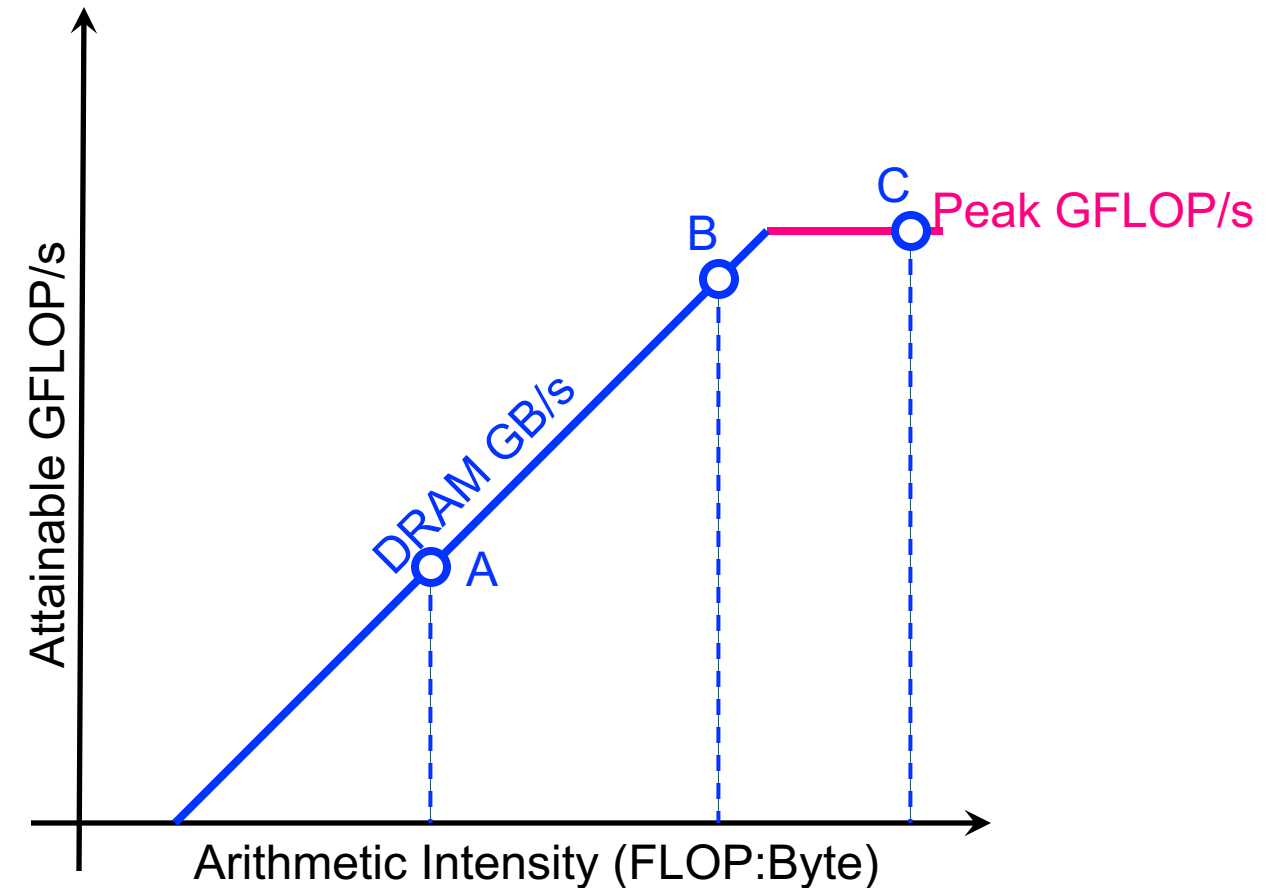


**Transition @ AI ==
Peak Gflop/s / Peak GB/s ==
'Machine Balance'**

Example



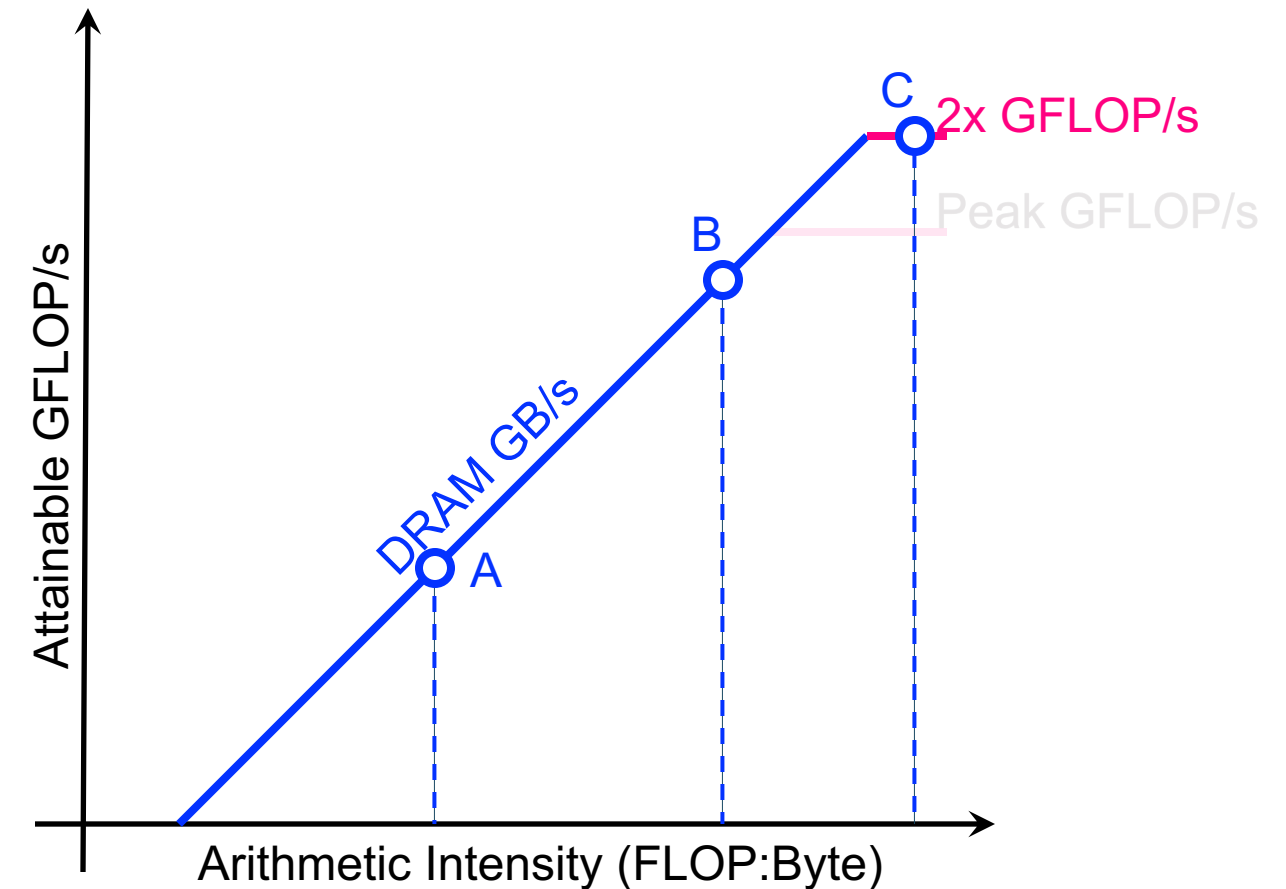
- Consider 3 kernels (A,B,C)
 - calculate or measure the **Arithmetic Intensity** for each
 - Determine the Roofline intercept for each kernel
 - **kernels A and B are bound by memory bandwidth**
 - **kernel C is bound by peak FLOP/s**



Scaling to Future GPUs



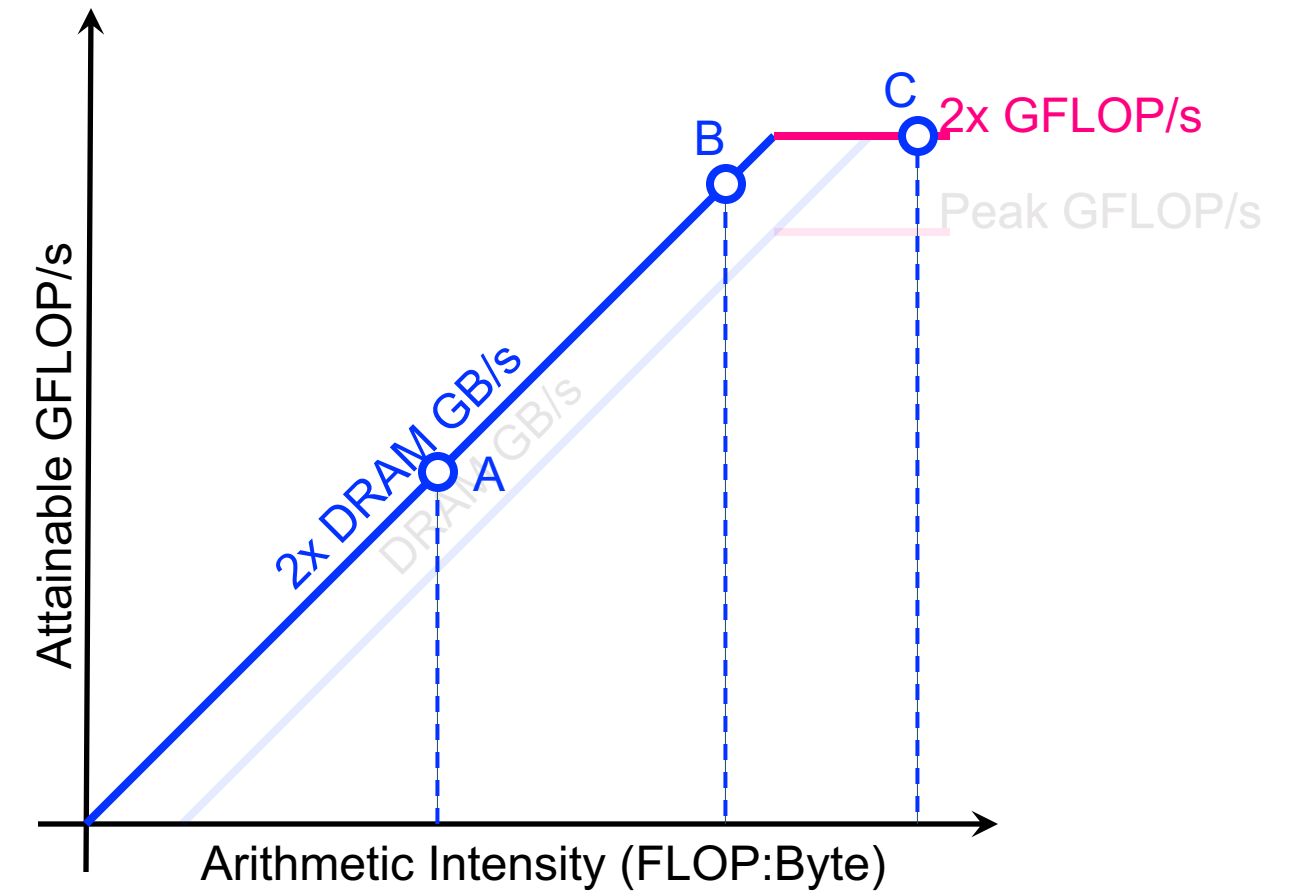
- Imagine you run on a future GPU with twice the peak FLOPs...
 - kernel C's performance could double
 - ✗ kernels A and B will be no faster



Scaling to Future GPUs



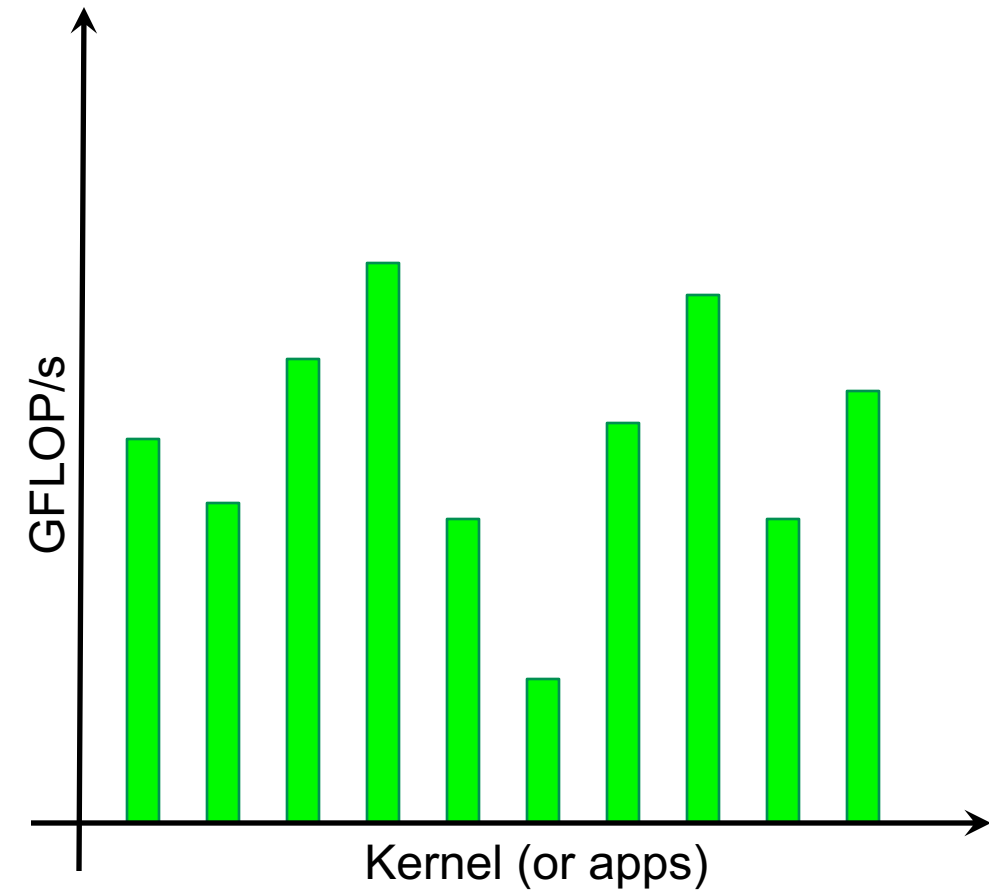
- What if that future GPU also doubled its memory bandwidth...
 - **kernel A and B's performance could also double**



Why is Roofline Useful?



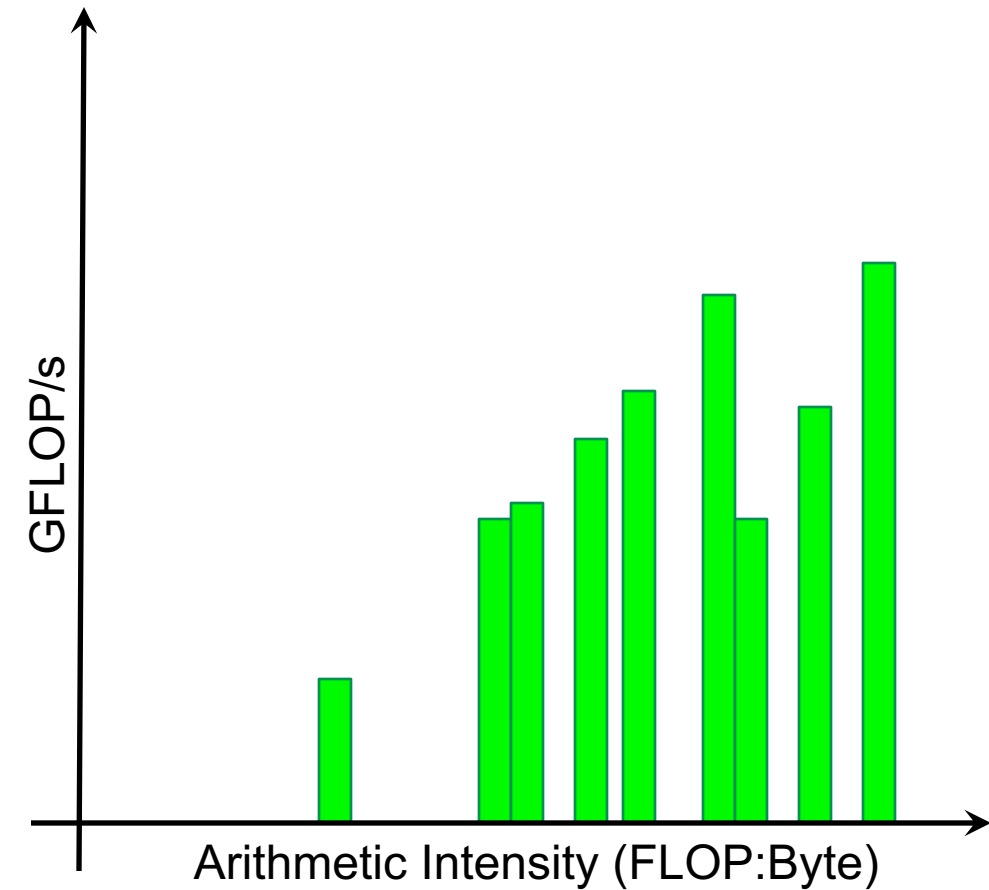
- Think back to our mix of loop nests where GFLOP/s alone wasn't useful...



Why is Roofline Useful?



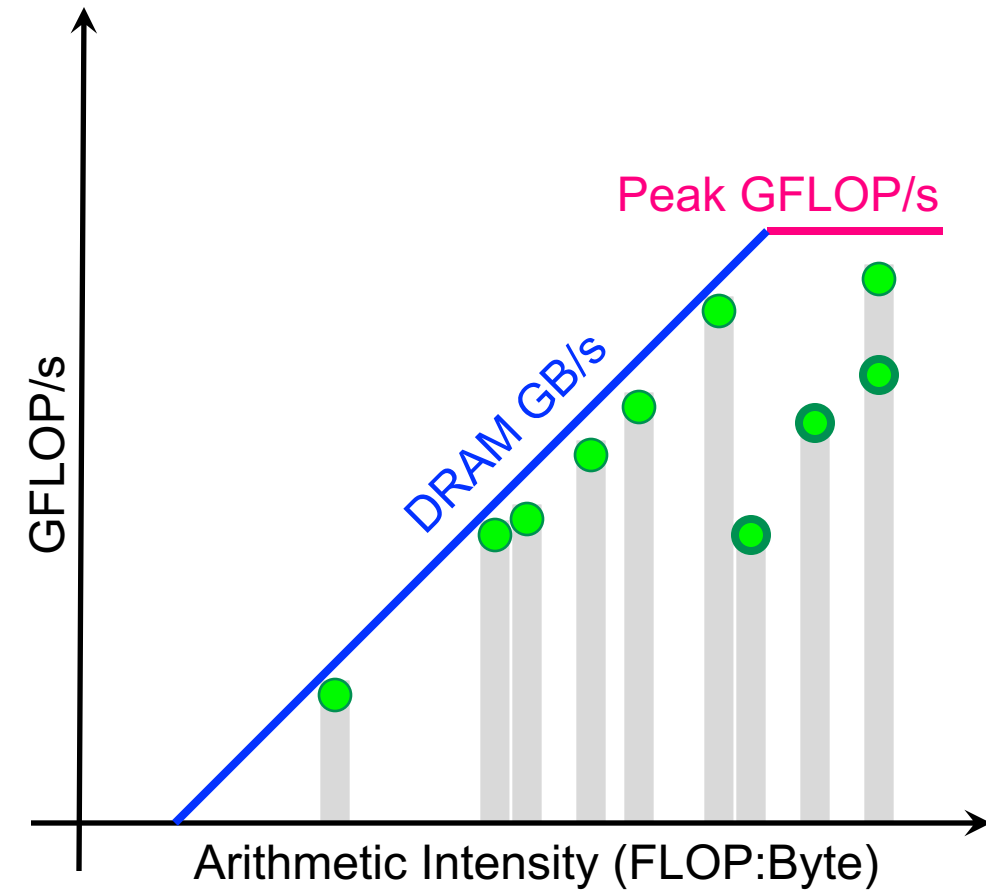
- We can sort kernels by AI ...



Why is Roofline Useful?

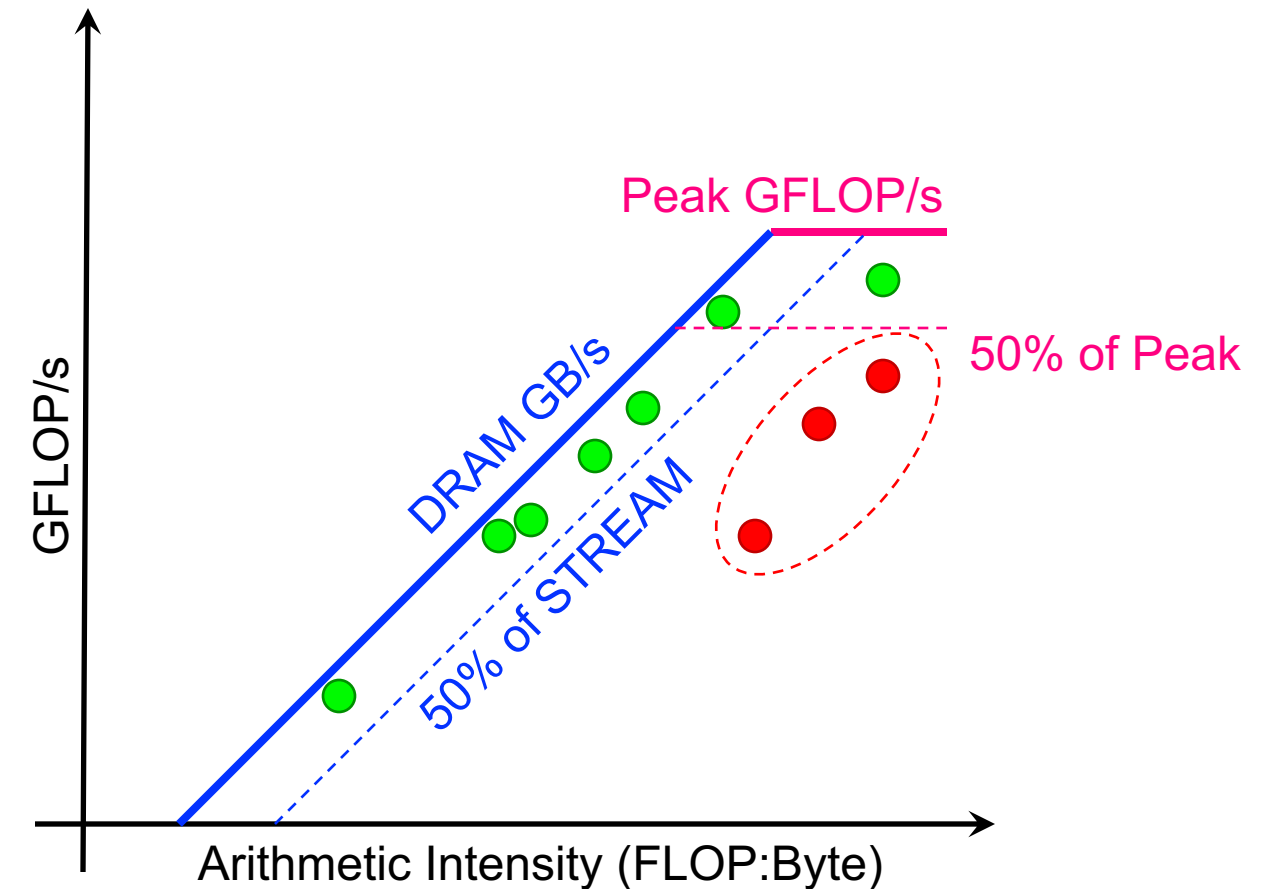


- We can sort kernels by AI ...
- ... and compare performance relative to machine capabilities



Why is Roofline Useful?

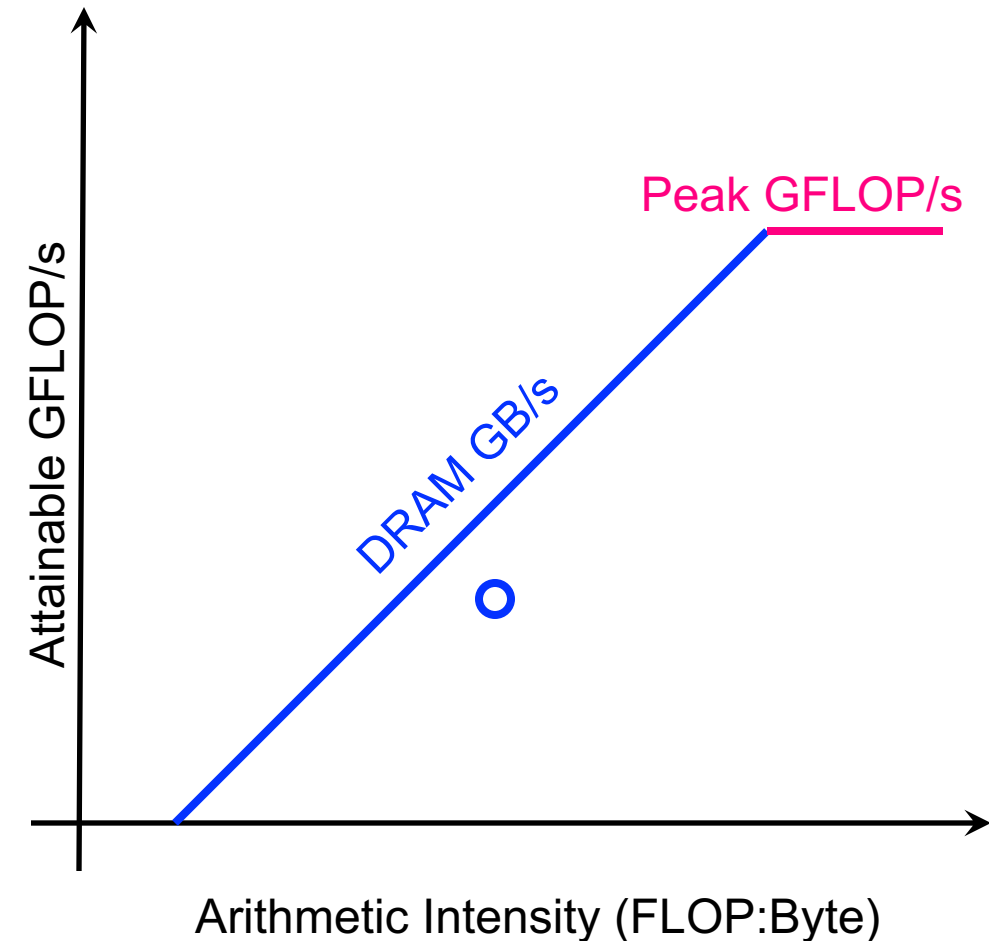
- Kernels near the roofline are making good use of computational resources...
 - kernels can have low performance (GFLOP/s), but make good use of a machine
 - kernels can have high performance (GFLOP/s), but make poor use of a machine



Can Performance Be Below Roofline?



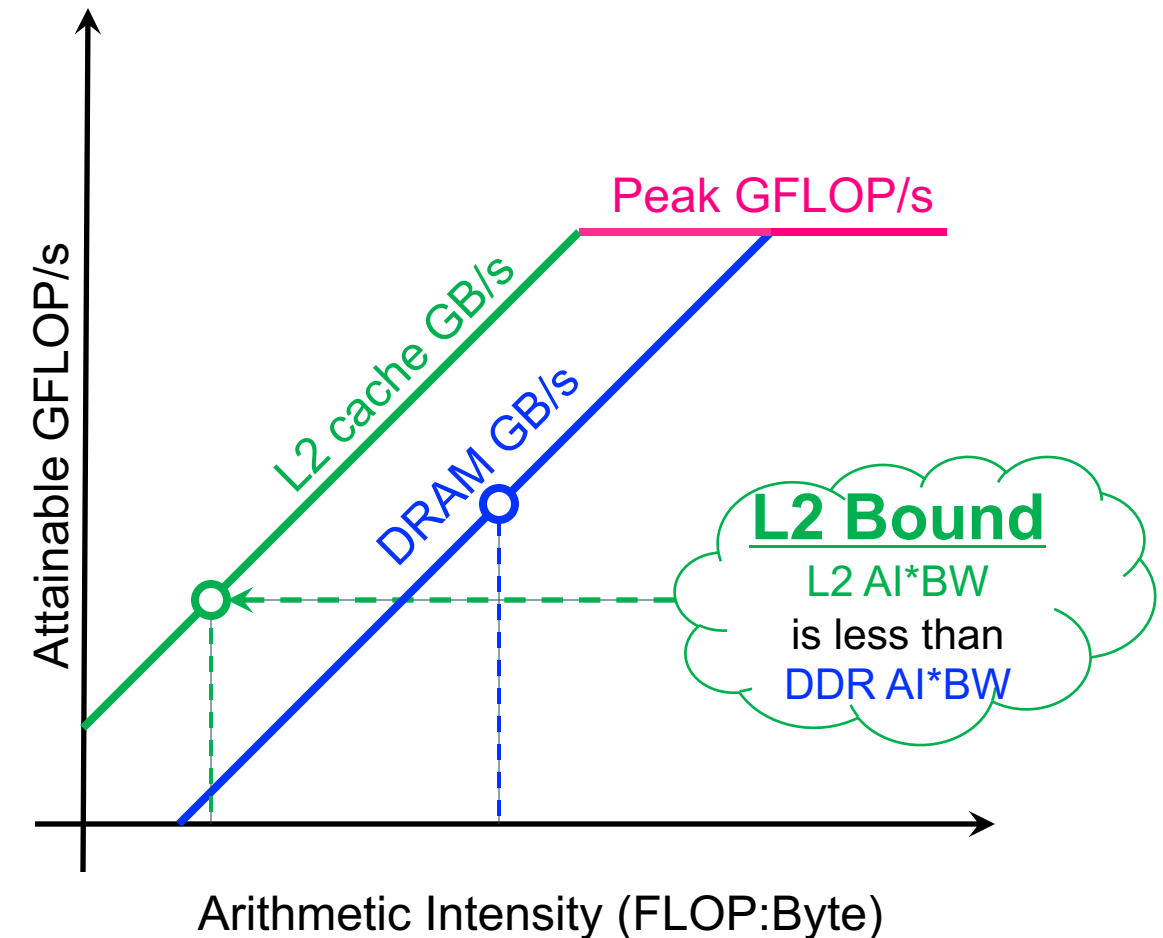
- Analogous to asking whether one can always attain either...
 - Peak Bandwidth
 - Peak GFLOP/s
- **Sure, there can be other performance bottlenecks...**
 - Cache bandwidth / locality
 - Lack of FMA / tensor instructions
 - Thread divergence / predication
 - Too many non-FP instructions
 - ...



Cache Effects...



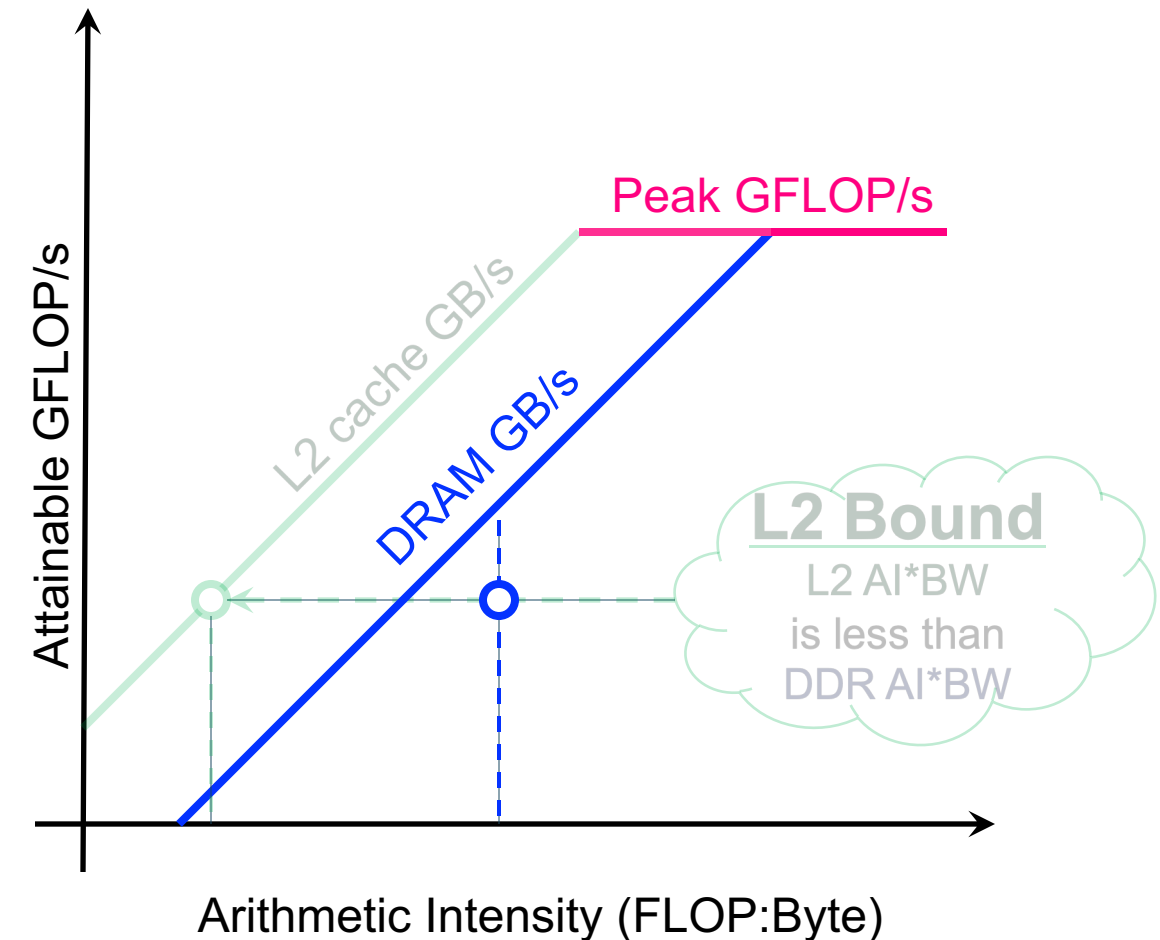
- Hierarchical Roofline Model
- Construct superposition of Rooflines...
 - Measure AI and bandwidth for each level of memory/cache
 - Loop nests will have multiple AI's and multiple performance bounds...
 - ... but performance is ultimately the minimum of these bounds.



Cache Effects...



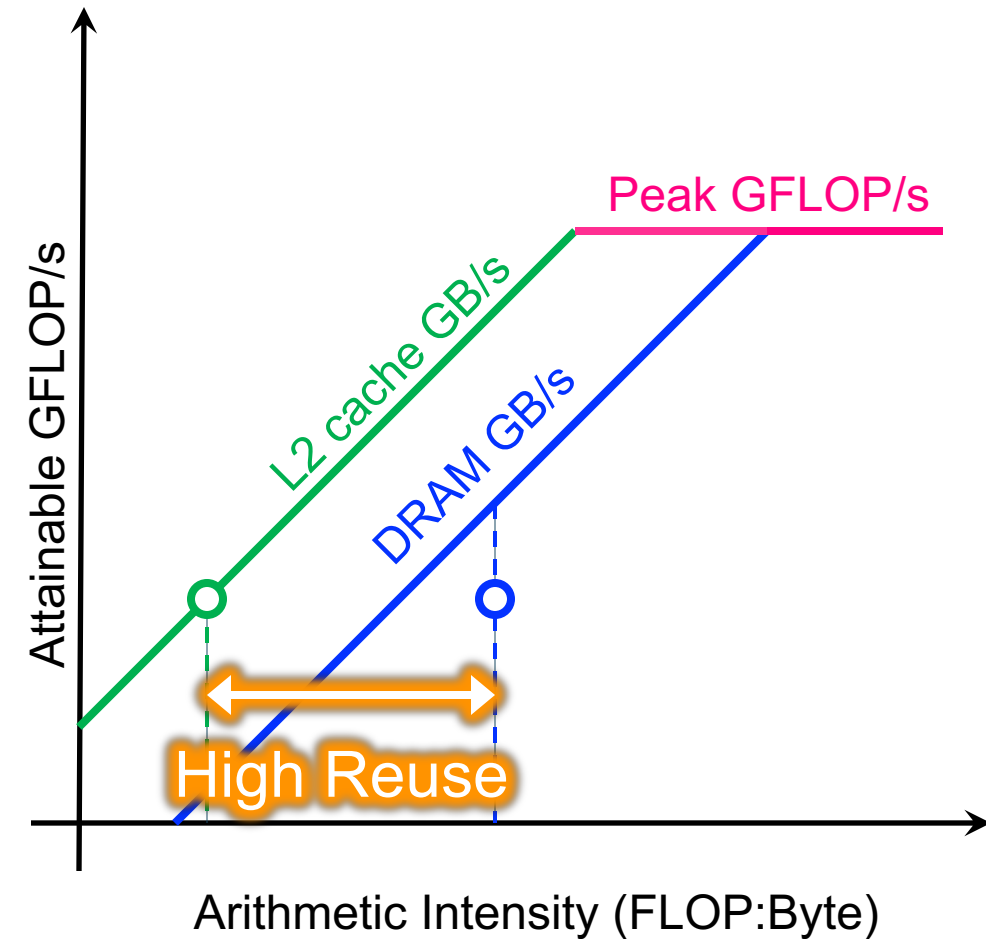
- Hierarchical Roofline Model
- Construct superposition of Rooflines...
 - Measure AI and bandwidth for each level of memory/cache
 - Loop nests will have multiple AI's and multiple performance bounds...
 - ... but performance is ultimately the minimum of these bounds.
- Extend to other memories...
 - L1 / Shared
 - System



Insights – Exploiting Caches



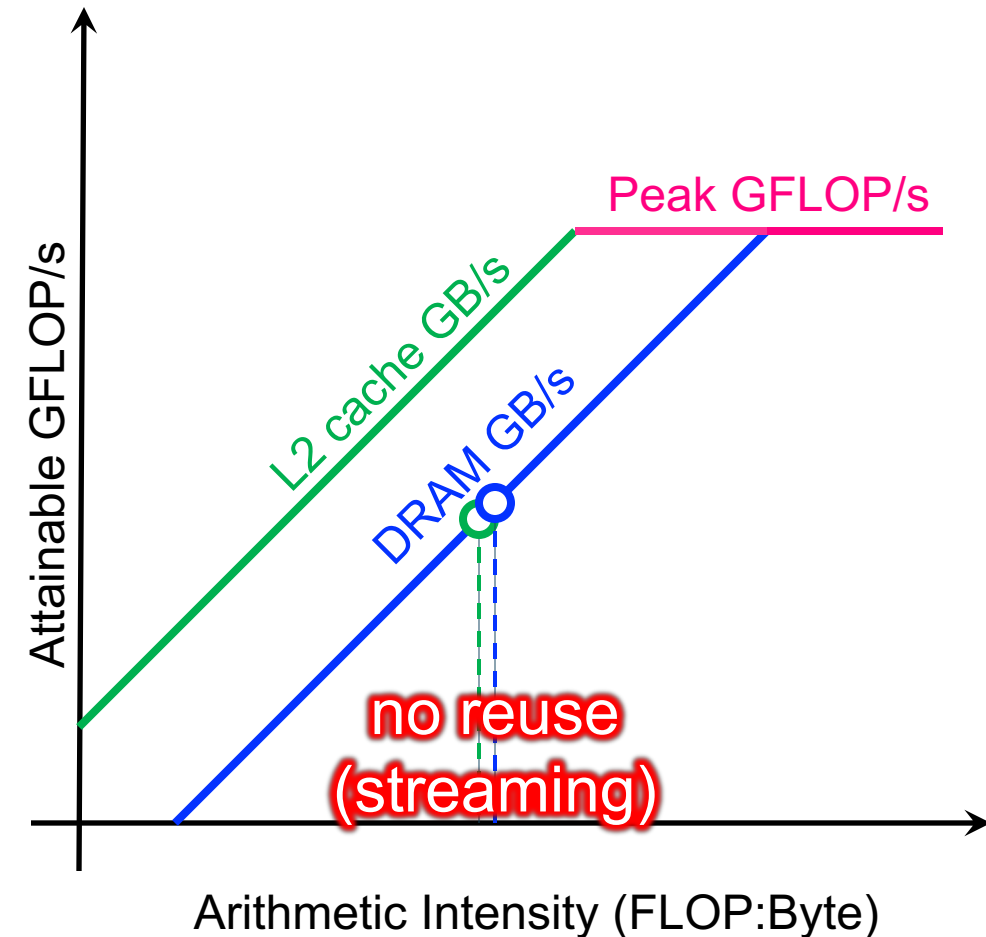
- Widely separated Arithmetic Intensities indicate high reuse in the cache



Insights – Exploiting Caches



- Widely separated Arithmetic Intensities indicate high reuse in the cache
- Similar Arithmetic Intensities indicate effectively no cache reuse (**== streaming**)
- As one changes problem size, L2 and DRAM arithmetic intensities can behave very differently



Failure to Exploit CISC Instructions

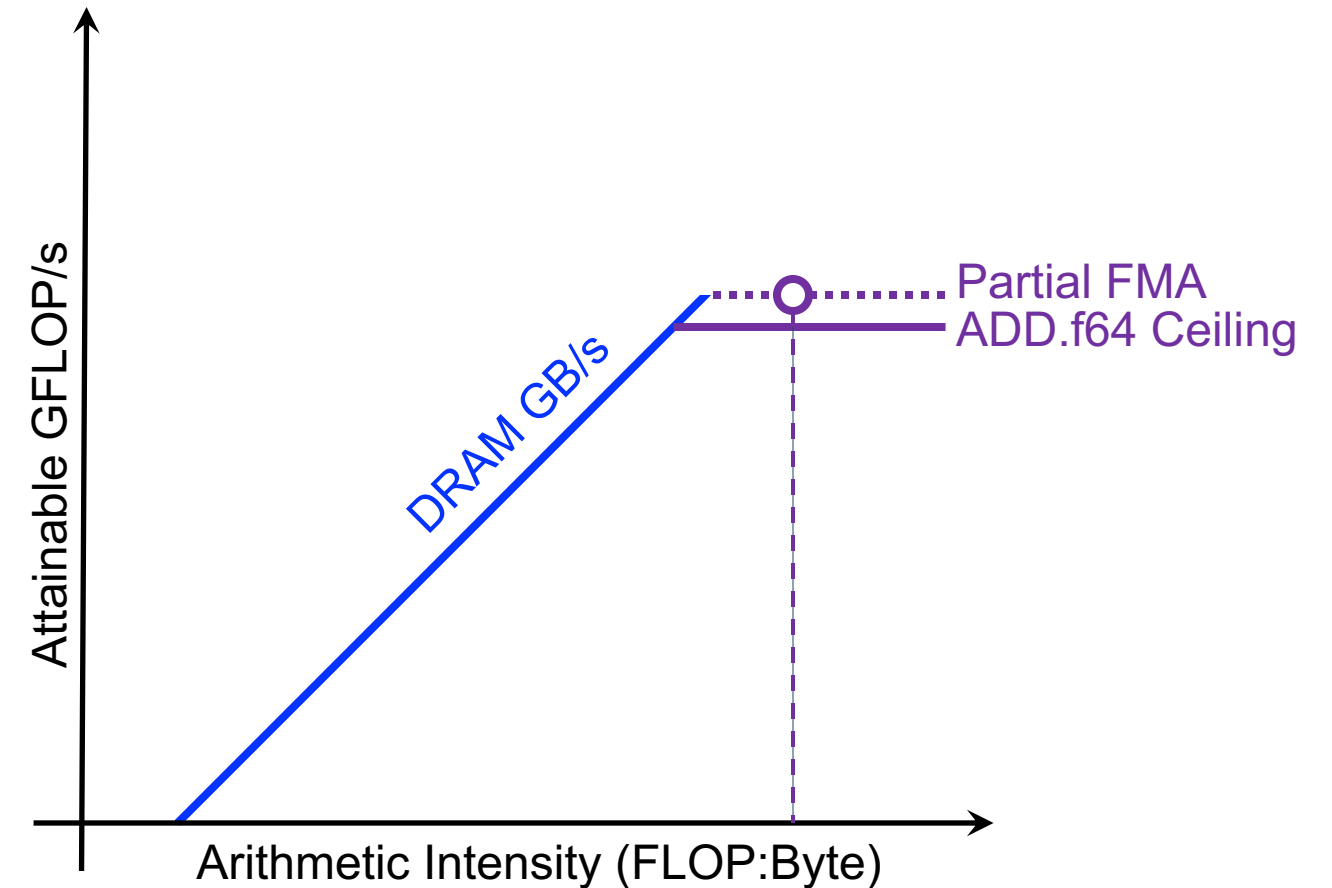


- Death of Moore's Law is motivating a return of Complex Instruction Set Computing (CISC)
- Modern CPUs and GPUs are increasingly reliant on special (fused) instructions that perform multiple operations.
 - FMA (Fused Multiply Add): $z=a*x+y$...*z,x,y are vectors or scalars*
 - 4FMA (quad FMA): $z=A*x+z$...*A is a FP32 matrix; x,z are vectors*
 - HMMA (Tensor Core): $Z=AB+C$...*Z,A,B,C are FP16 matrices*
 - ...
- **Performance is now a weighted average of Mul/Add, FMA, and HMMA operations.**

Failure to Exploit CISC Instructions



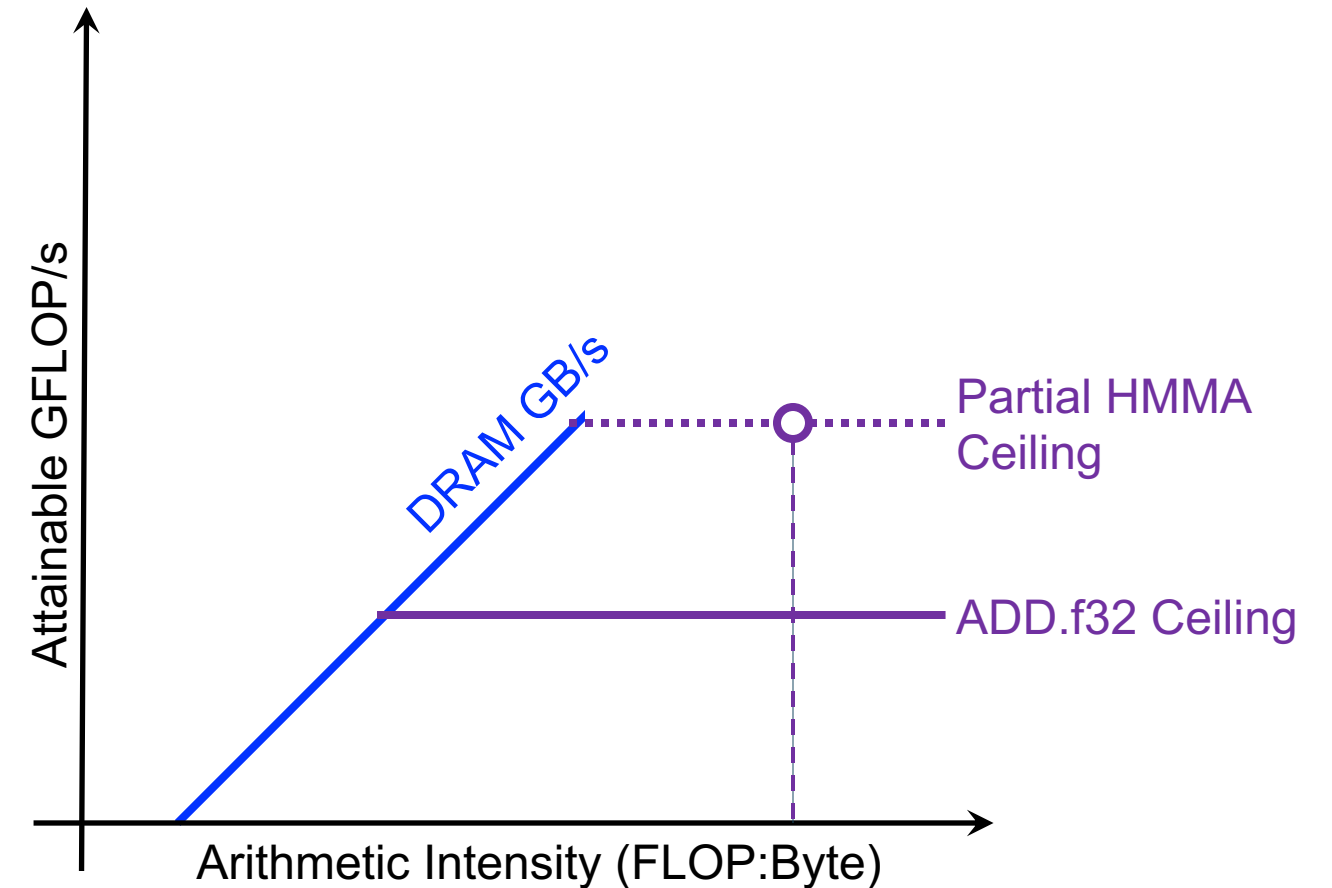
- Total lack of FMA reduces Volta performance by 2x...
 - **creates ADD.f64 ceiling**
- In reality, applications are a mix of FMA.f64, ADD.f64, and MUL.f64...
 - Performance is a weighted average
 - **Produces a partial FMA ceiling that bounds kernel performance**

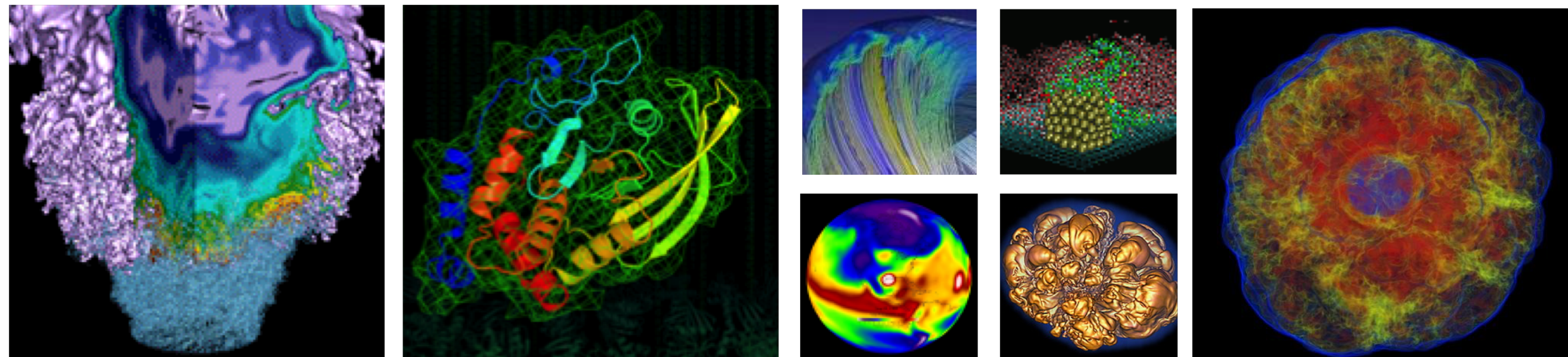


Failure to Exploit CISC Instructions



- On Volta, Tensor cores provide **125 TFLOPs** of FP16 performance (vs. 15 for FP32)
- However, kernels/apps will mix HMMA with FMA, MULs, ADDs, ...
 - **A few non-HMMA operations can quickly limit Tensor core performance**





Using Roofline To Drive Optimization



U.S. DEPARTMENT OF
ENERGY

Office of
Science

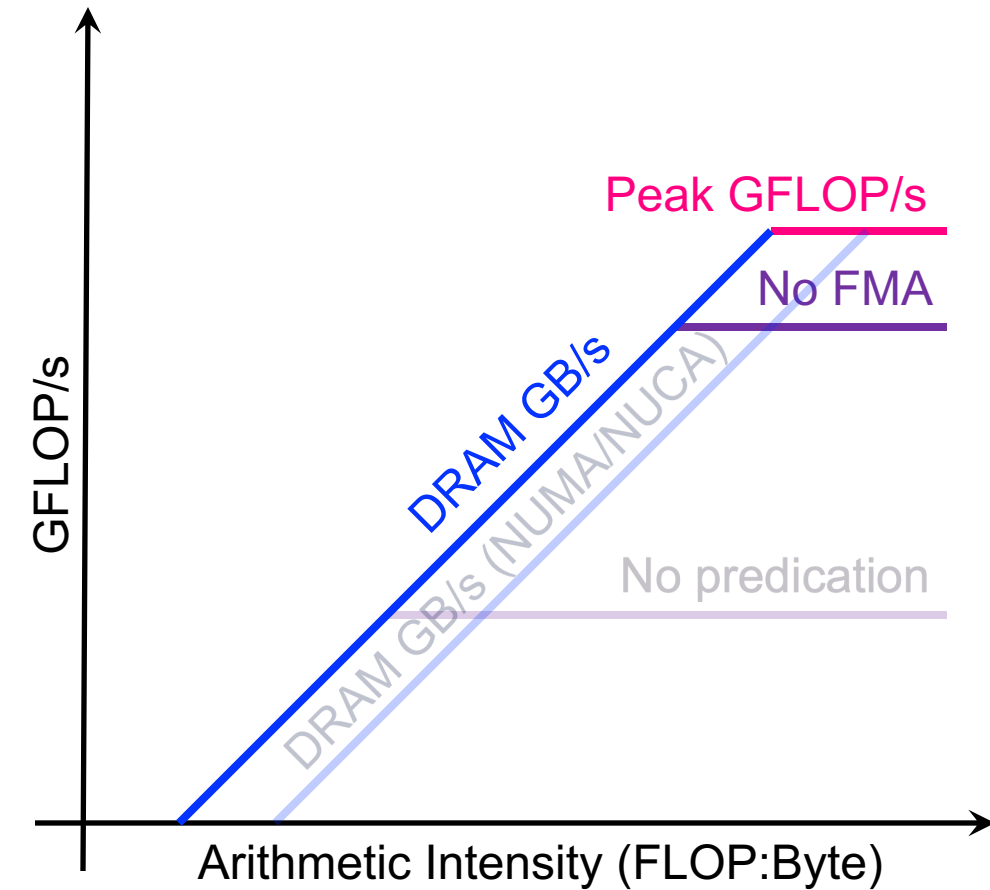


BERKELEY LAB
Lawrence Berkeley National Laboratory

Driving Performance Optimization



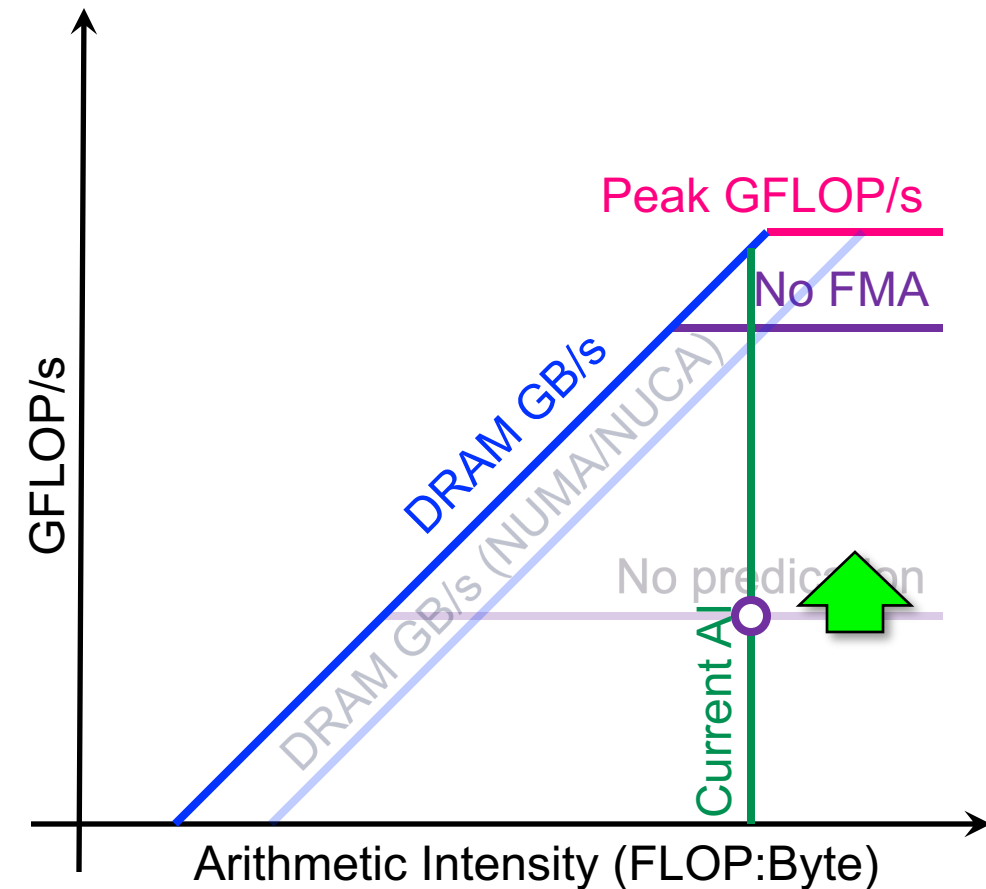
- Broadly speaking, there are three approaches to improving performance:



Driving Performance Optimization



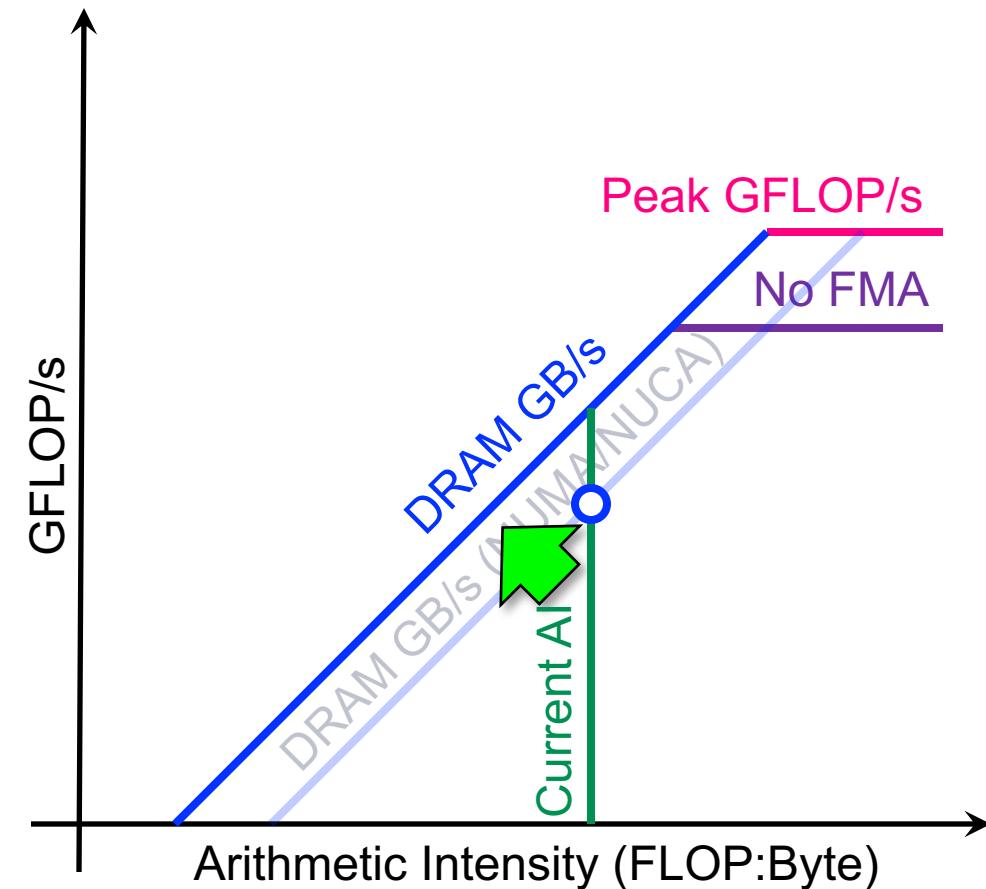
- Broadly speaking, there are three approaches to improving performance:
- **Maximize SM performance (e.g. minimize predication)**



Driving Performance Optimization



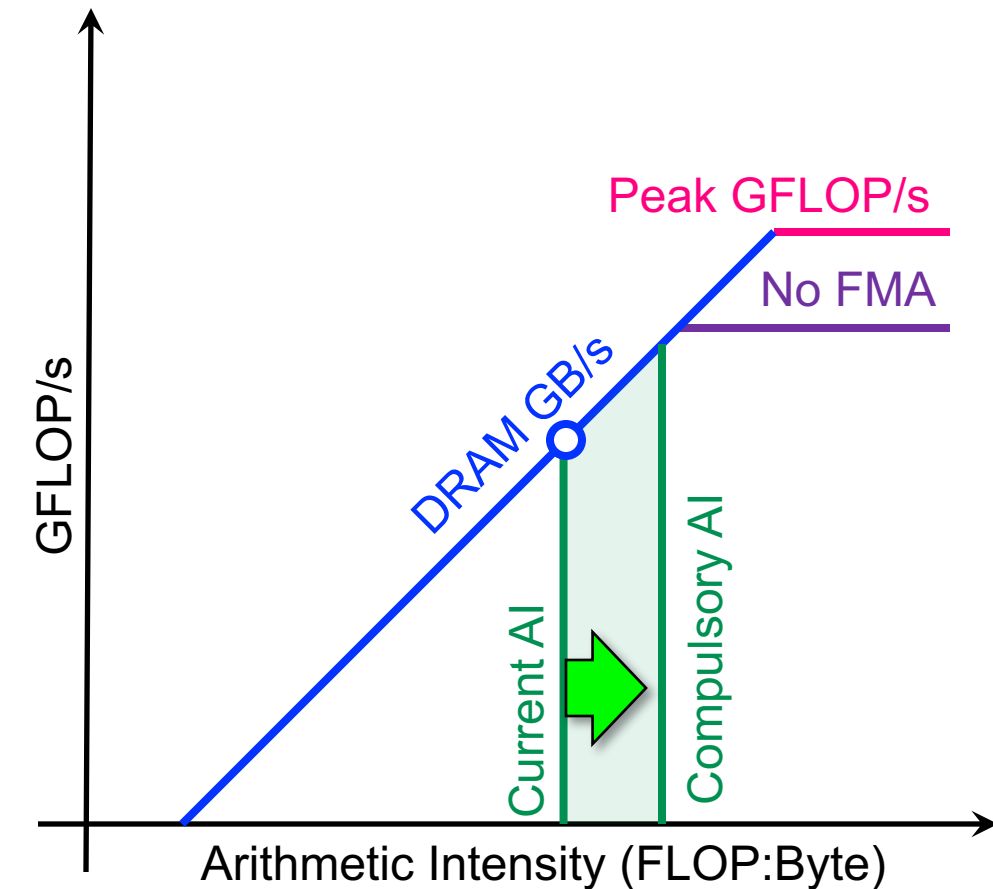
- Broadly speaking, there are three approaches to improving performance:
- Maximize SM performance (e.g. minimize predication)
- **Maximize memory bandwidth (e.g. avoid pathological memory access patterns)**

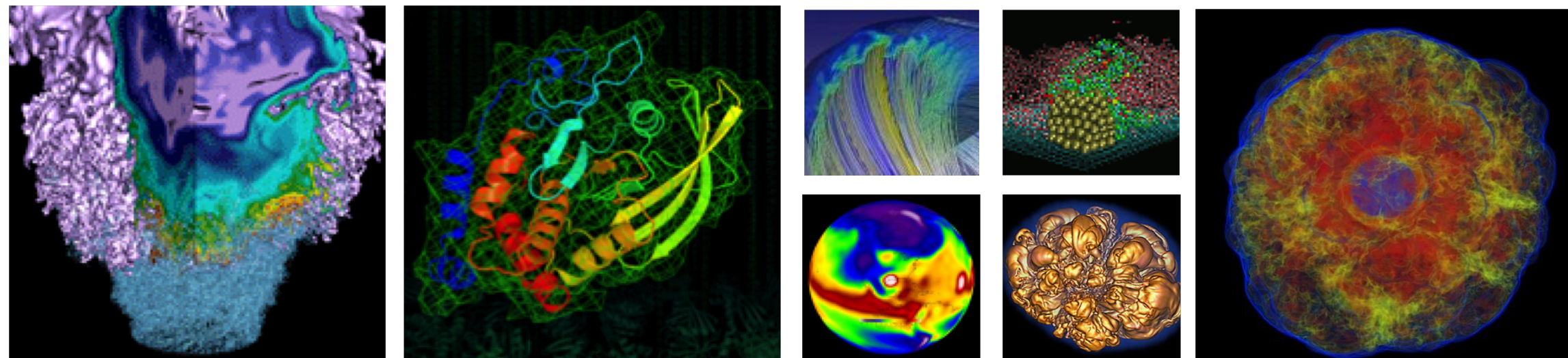


Driving Performance Optimization



- Broadly speaking, there are three approaches to improving performance:
- Maximize SM performance (e.g. minimize predication)
- Maximize memory bandwidth (e.g. avoid pathological memory access patterns)
- **Minimize data movement (i.e. exploit reuse)**





Estimating Arithmetic Intensity



U.S. DEPARTMENT OF
ENERGY

Office of
Science



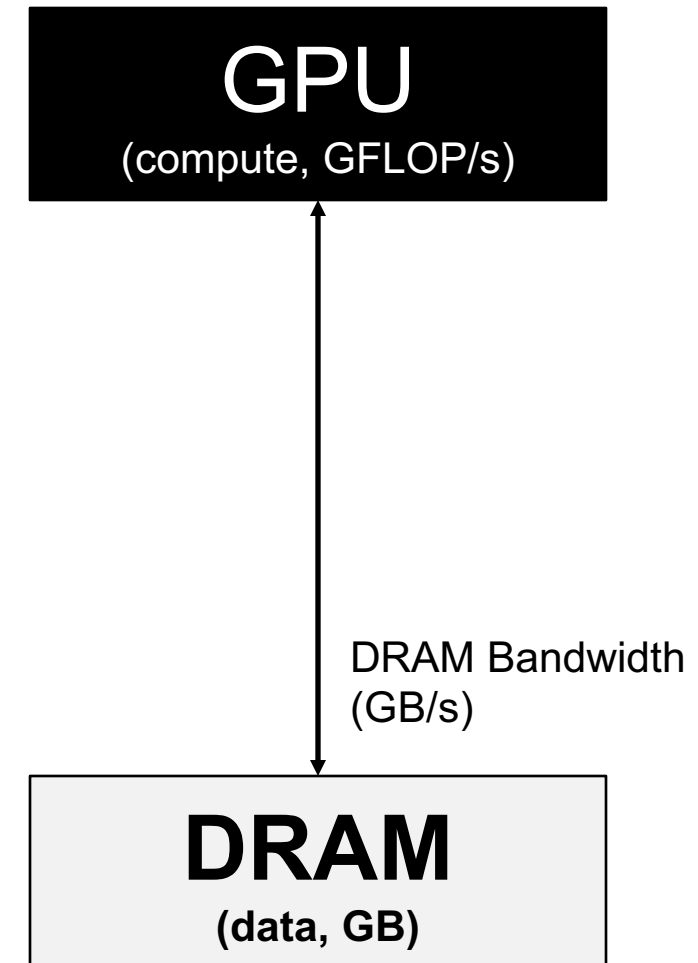
BERKELEY LAB
Lawrence Berkeley National Laboratory

DRAM vs L1 Arithmetic Intensity



- Consider a 7-point constant coefficient stencil...
 - 7 FLOPs
 - 8 memory references (7 reads, 1 store) per point
 - **AI = 0.11 FLOPs per byte (L1)**

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
    new[k][j][i] = -6.0*old[k ][j ][i ]
                  + old[k ][j ][i-1]
                  + old[k ][j ][i+1]
                  + old[k ][j-1][i ]
                  + old[k ][j+1][i ]
                  + old[k-1][j ][i ]
                  + old[k+1][j ][i ];
}}}
```

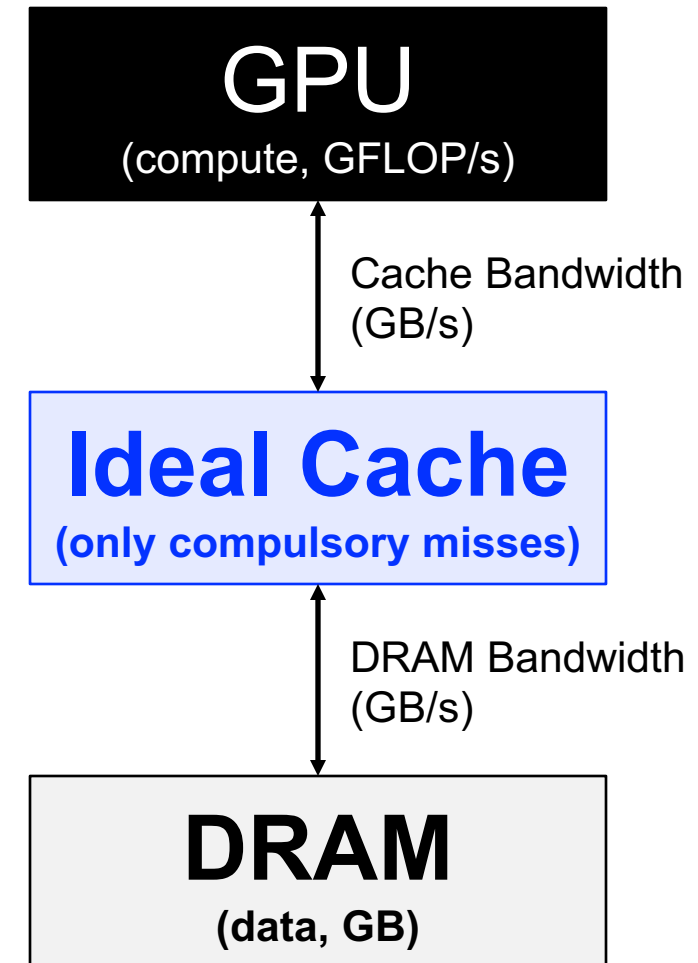


DRAM vs L1 Arithmetic Intensity



- Consider a 7-point constant coefficient stencil...
 - 7 FLOPs
 - 8 memory references (7 reads, 1 store) per point
 - Cache can filter all but 1 read and 1 write per point
 - **AI = 0.44 FLOPs per byte**

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
    new[k][j][i] = -6.0*old[k ][j ][i ]
                    + old[k ][j ][i-1]
                    + old[k ][j ][i+1]
                    + old[k ][j-1][i ]
                    + old[k ][j+1][i ]
                    + old[k-1][j ][i ]
                    + old[k+1][j ][i ]
}}}
```

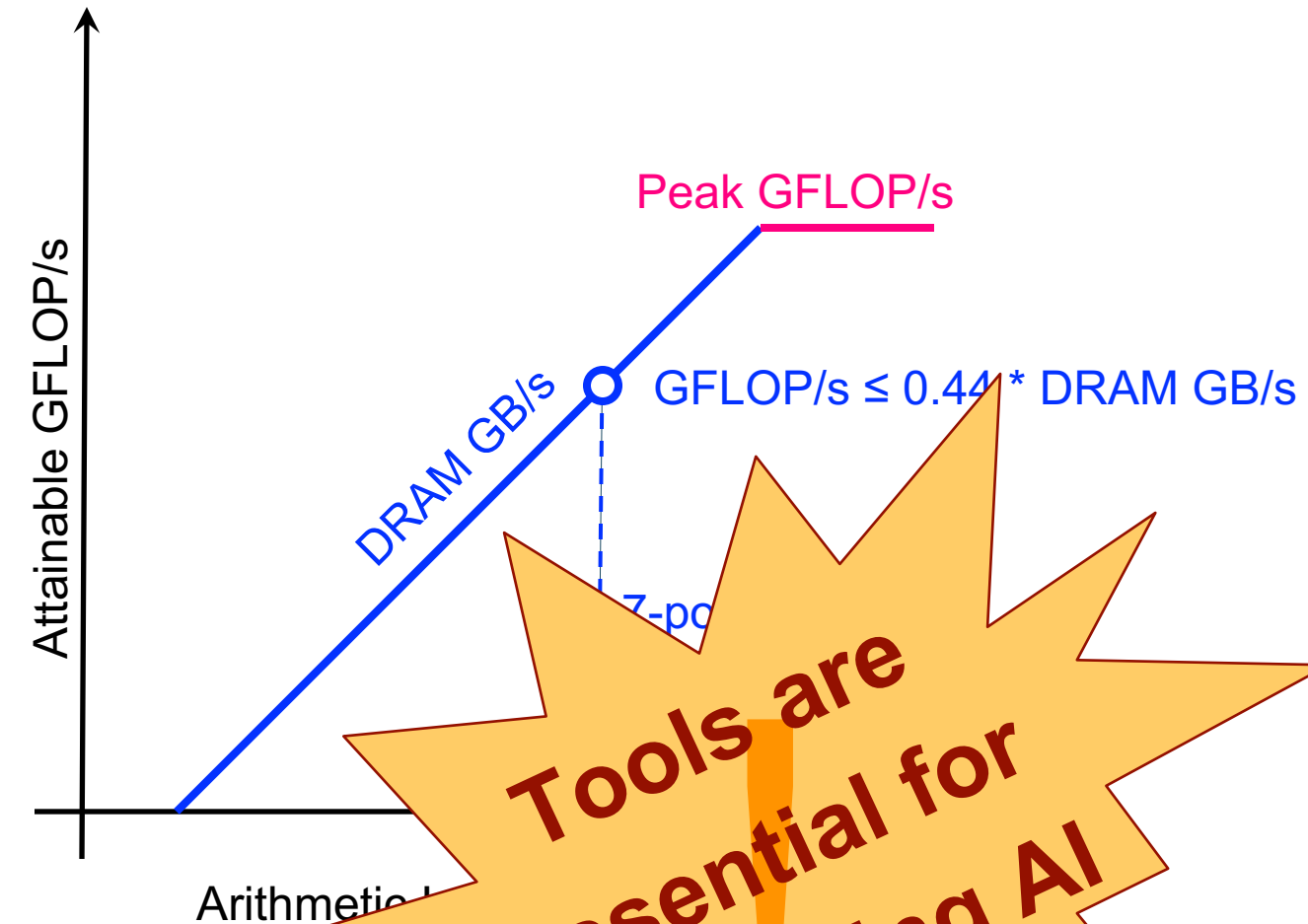


DRAM vs L1 Arithmetic Intensity

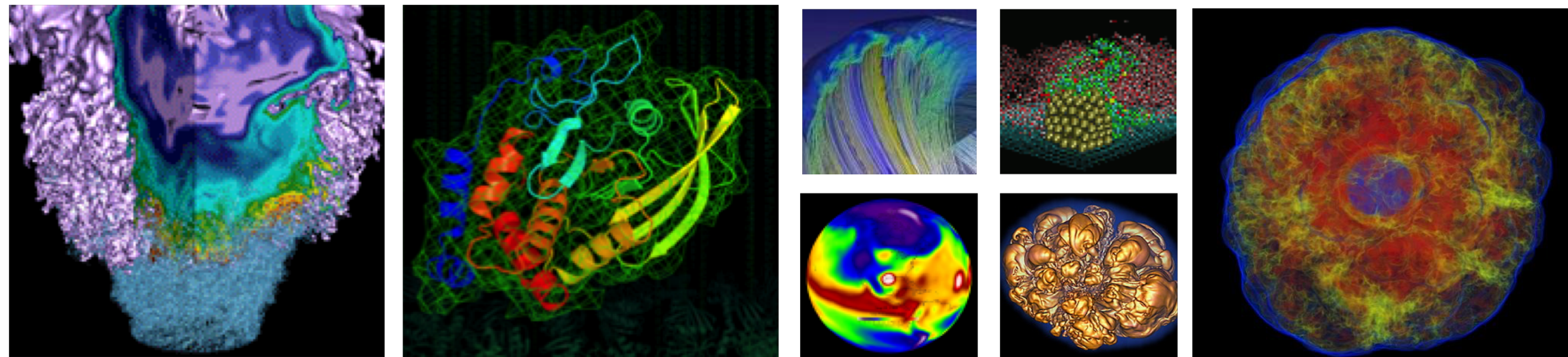


- Consider a 7-point constant coefficient stencil...
 - 7 FLOPs
 - 8 memory references (7 reads, 1 store) per point
 - Cache can filter all but 1 read and 1 write per point
 - **AI = 0.44 FLOPs per byte == memory bound**

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
new[k][j][i] = -6.0*old[k ][j ][i ]
                + old[k ][j ][i-1]
                + old[k ][j ][i+1]
                + old[k ][j-1][i ]
                + old[k ][j+1][i ]
                + old[k-1][j ][i ]
                + old[k+1][j ][i ];
}}}
```



Tools are essential for measuring AI



Collecting Roofline Data with nvprof



U.S. DEPARTMENT OF
ENERGY

Office of
Science



BERKELEY LAB
Lawrence Berkeley National Laboratory

General Roofline Data Collection



Most kernels are more complicated than the 7-point stencil...

General Roofline Data Collection



Most kernels are more complicated than the 7-point stencil...

How do we measure the total number of FLOPs?

How do we measure the total number of bytes moved (read/write, L1/L2/HBM)?

How do we measure the runtime for each kernel?

How do we know the peak bandwidth (L1/L2/HBM) and the peak FLOP/s for the architecture?

General Roofline Data Collection



Most kernels are more complicated than the 7-point stencil...

How do we measure the total number of FLOPs?

How do we measure the total number of bytes moved (read/write, L1/L2/HBM)?

How do we measure the runtime for each kernel?



How do we know the peak bandwidth (L1/L2/HBM) and the peak FLOP/s for the architecture?



Step 1. Collect Roofline Ceilings



- **Empirical Roofline Toolkit (ERT)**
 - Different than the architecture specs, **MORE REALISTIC**
 - Reflects **actual** execution environment (power constraints, *etc*)
 - Sweeps through a range of configurations, and **statistically stable**
 - Data elements per thread
 - FLOPs per data element
 - Threadblocks/threads
 - Trails per dataset
 - *etc*



Kernel.c

- actual compute
- customizable

Driver.c

- setup
- call kernels
- loop over parameters

config script

- set up ranges of parameters

job script

- submit the job and run it

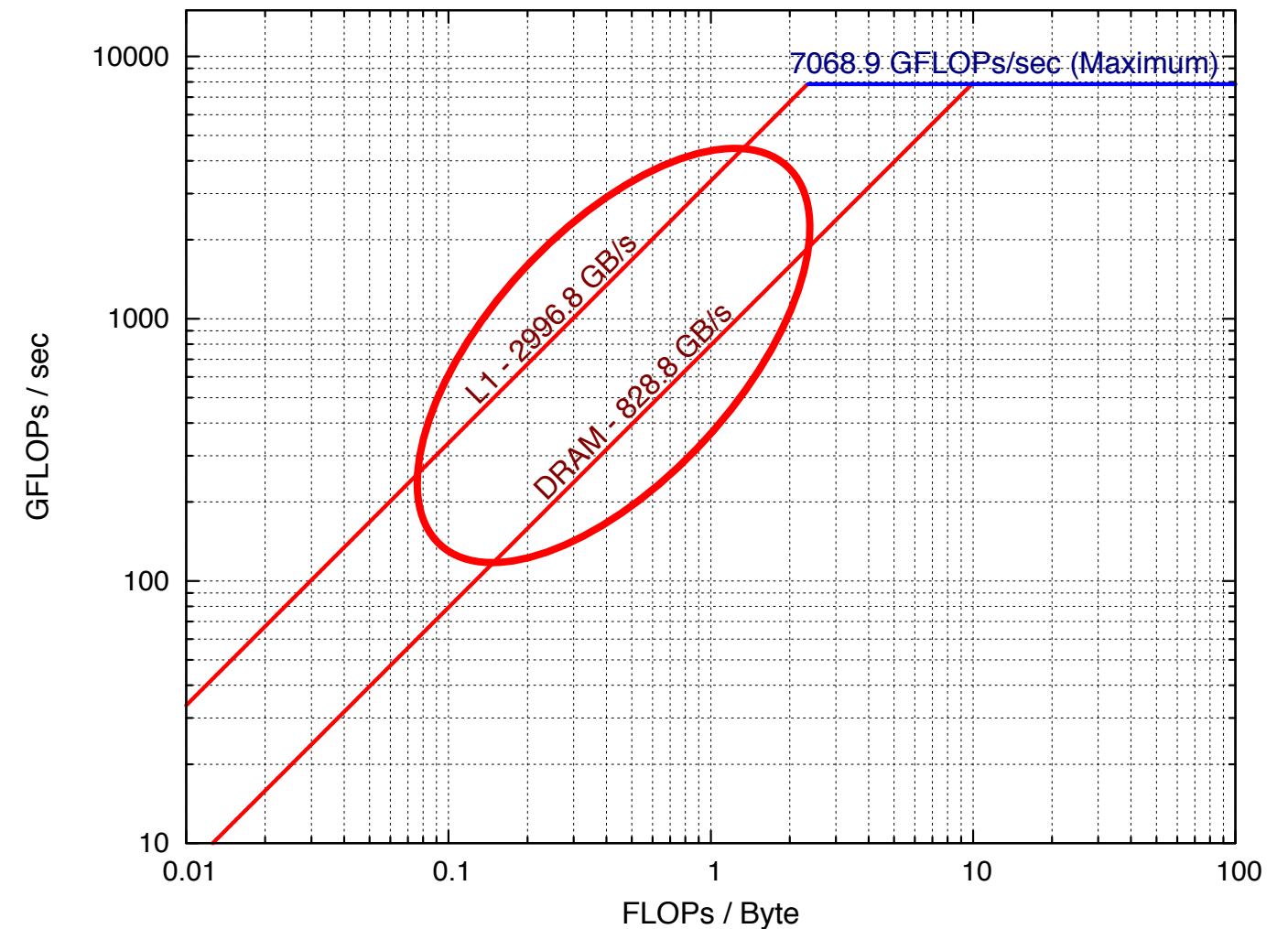
ERT Output



roofline.json

```
"gbytes": {  
  "data": [  
    [  
      "L1",  
      2996.82  
    ],  
    [  
      "DRAM",  
      828.83  
    ]  
  ],  
},  
  
"gflops": {  
  "data": [  
    [  
      "GFLOPs",  
      7068.90  
    ]  
  ],  
},
```

roofline.ps



ERT Output

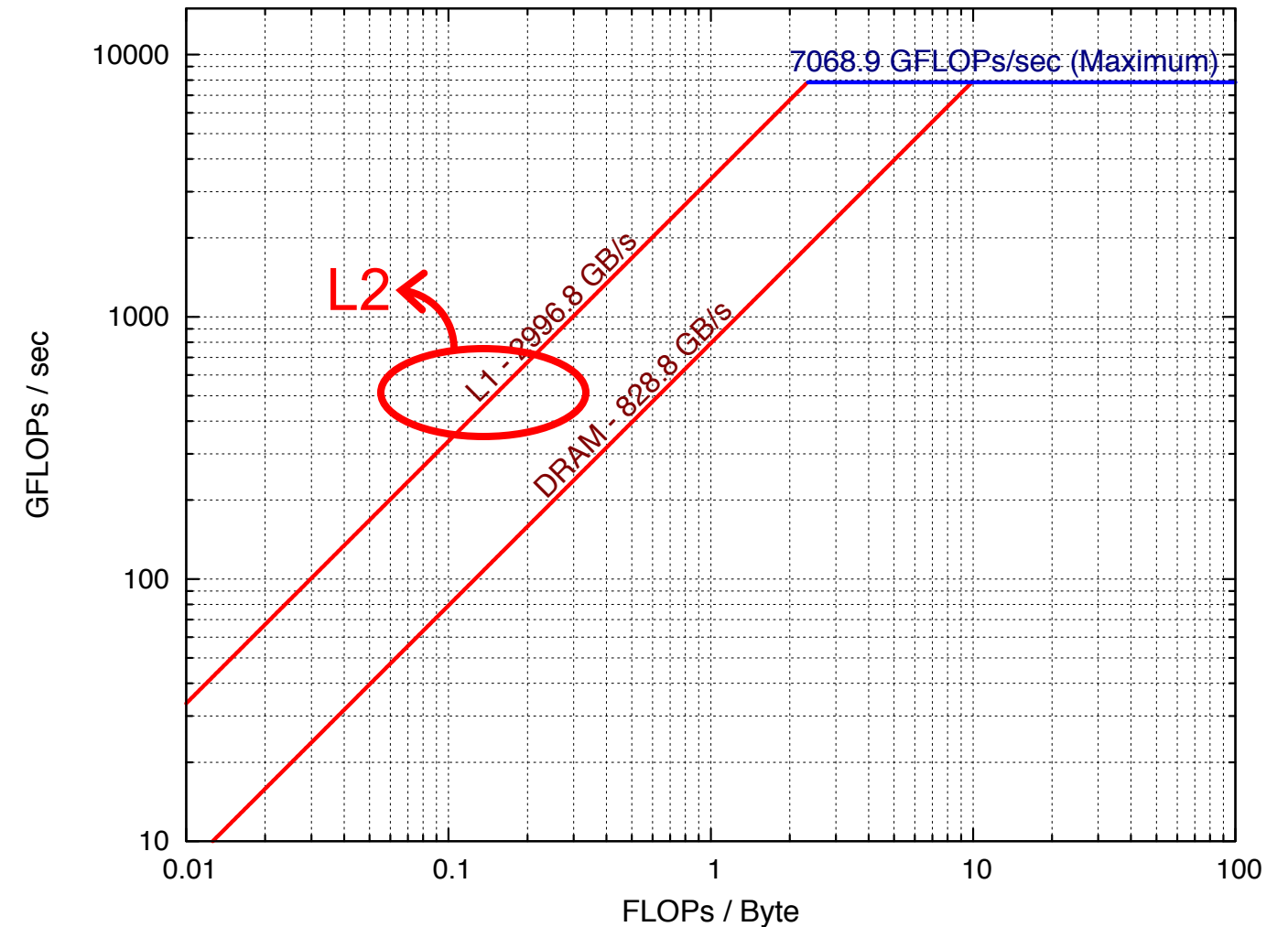


roofline.json

```
"gbytes": {  
  "data": [  
    [  
      "L1",  
      2996.82  
    ],  
    [  
      "DRAM",  
      828.83  
    ]  
  ],  
}
```

```
"gflops": {  
  "data": [  
    "GFLOPs",  
    7068.90  
  ]  
}
```

roofline.ps



ERT Output

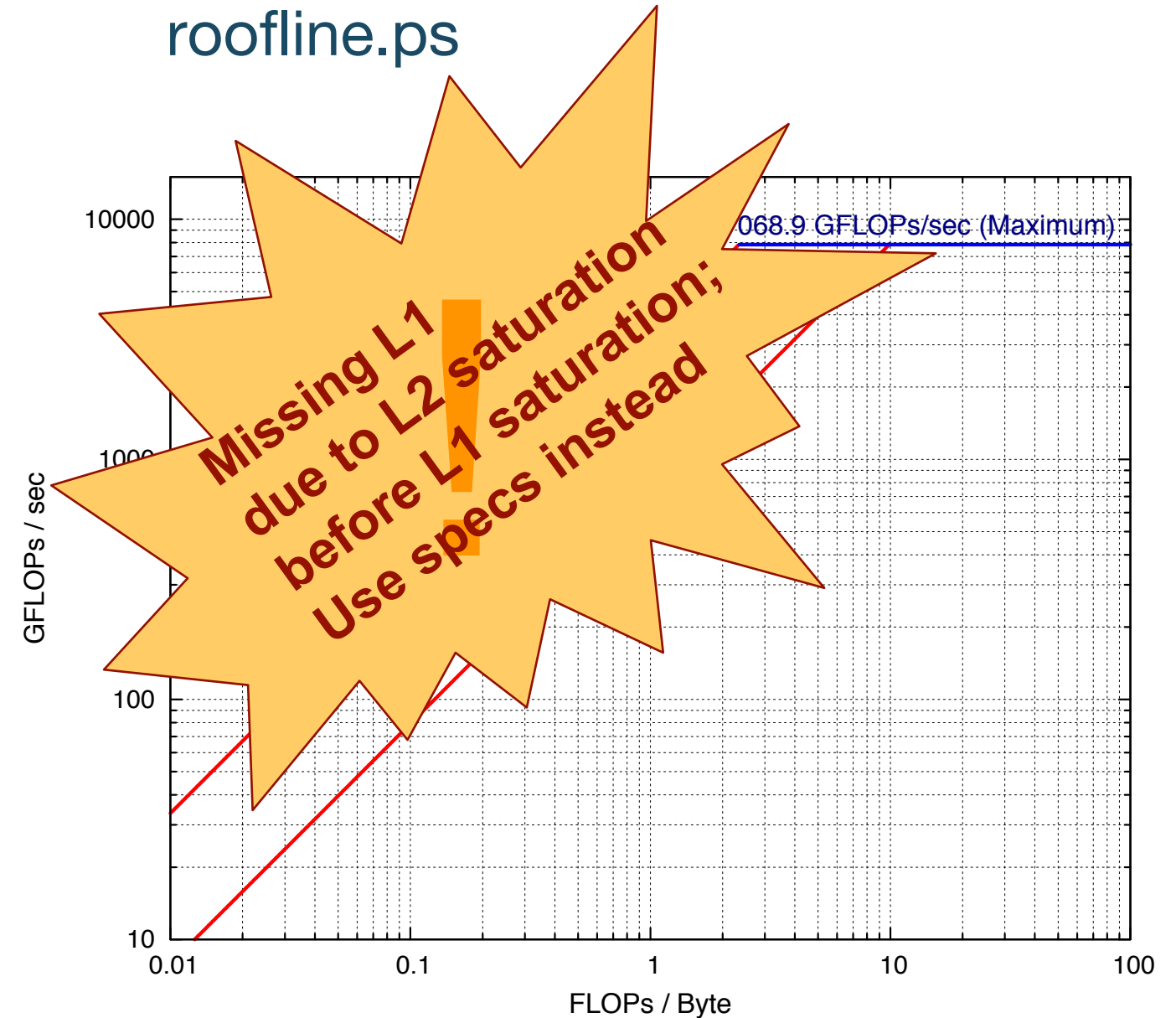


roofline.json

```
"gbytes": {  
  "data": [  
    [  
      "L1",  
      2996.82  
    ],  
    [  
      "DRAM",  
      828.83  
    ]  
  ],  
}
```

```
"gflops": {  
  "data": [  
    [  
      "GFLOPs",  
      7068.90  
    ]  
  ],  
}
```

roofline.ps



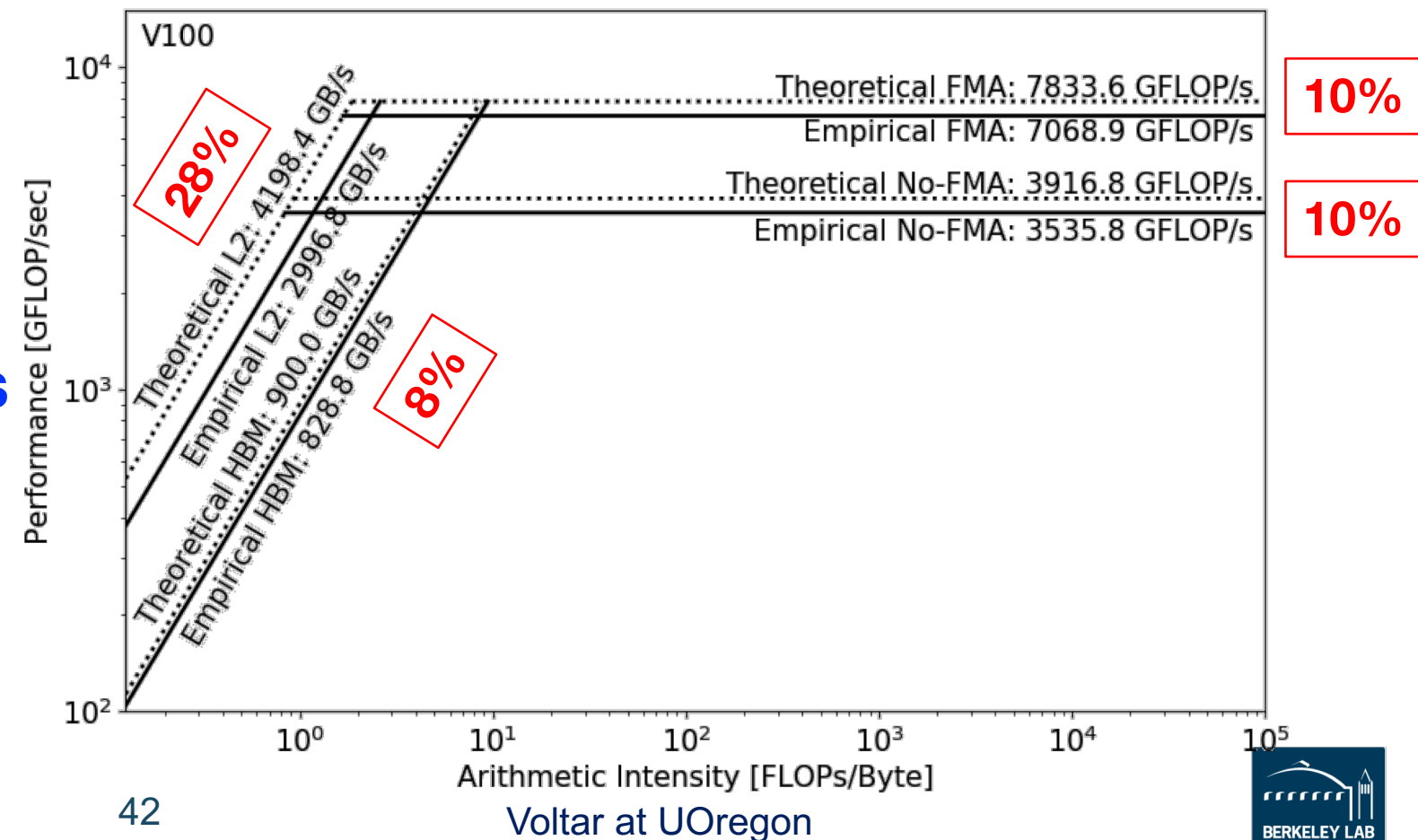
Discrepancy Empirical vs. Theoretical



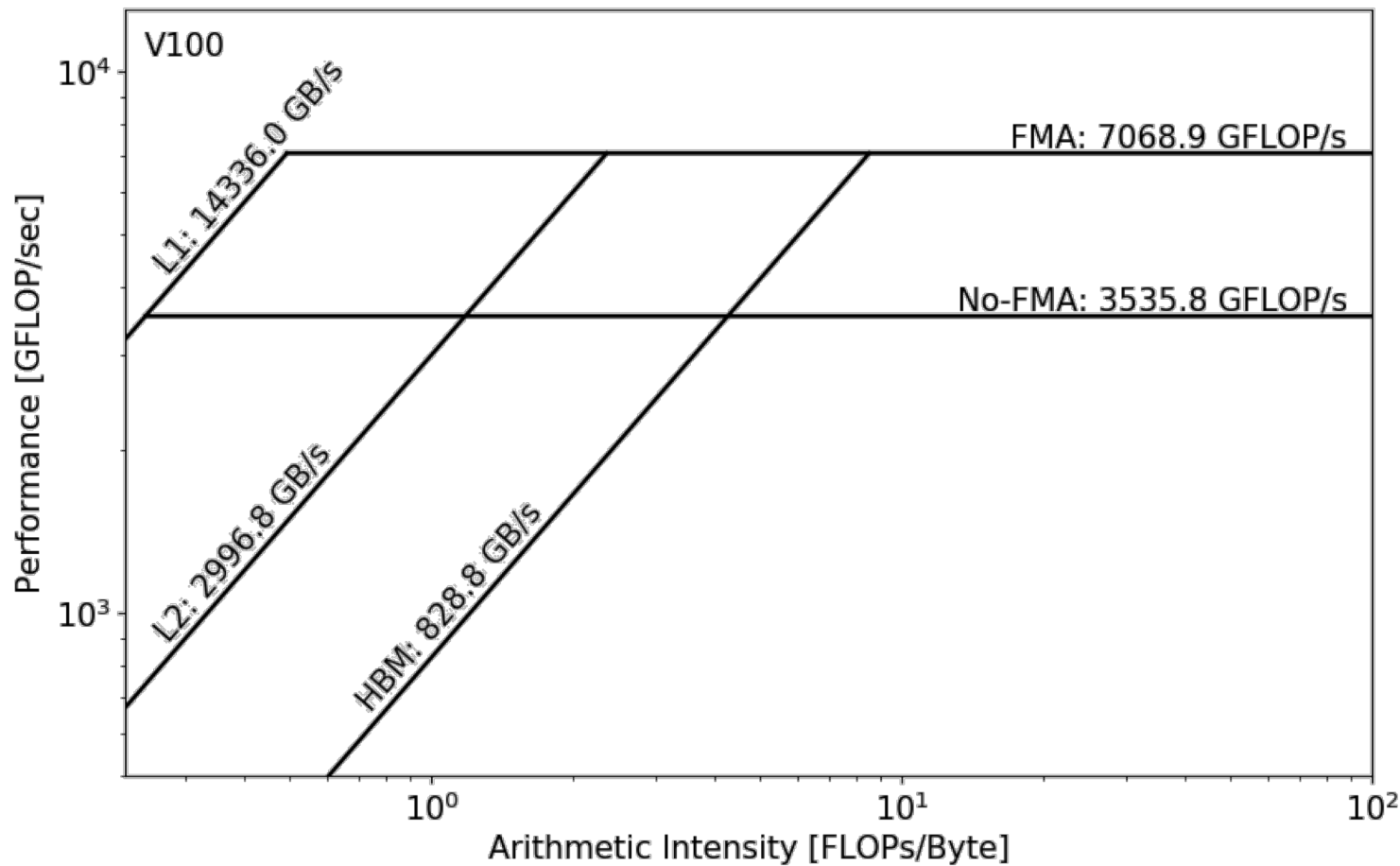
- Theoretical FP64 **compute** ceilings on V100:
 - FMA: 80 SMs x 32 FP64 cores x 1.53 GHz x 2 = 7.83 TFLOP/s
 - no FMA: 80 SMs x 32 FP64 cores x 1.53 GHz = 3.92 TFLOP/s
- Theoretical **memory** bandwidths on V100:

- HBM: 900 GB/s
- L2: ~4.1 TB/s
- L1: ~14 TB/s

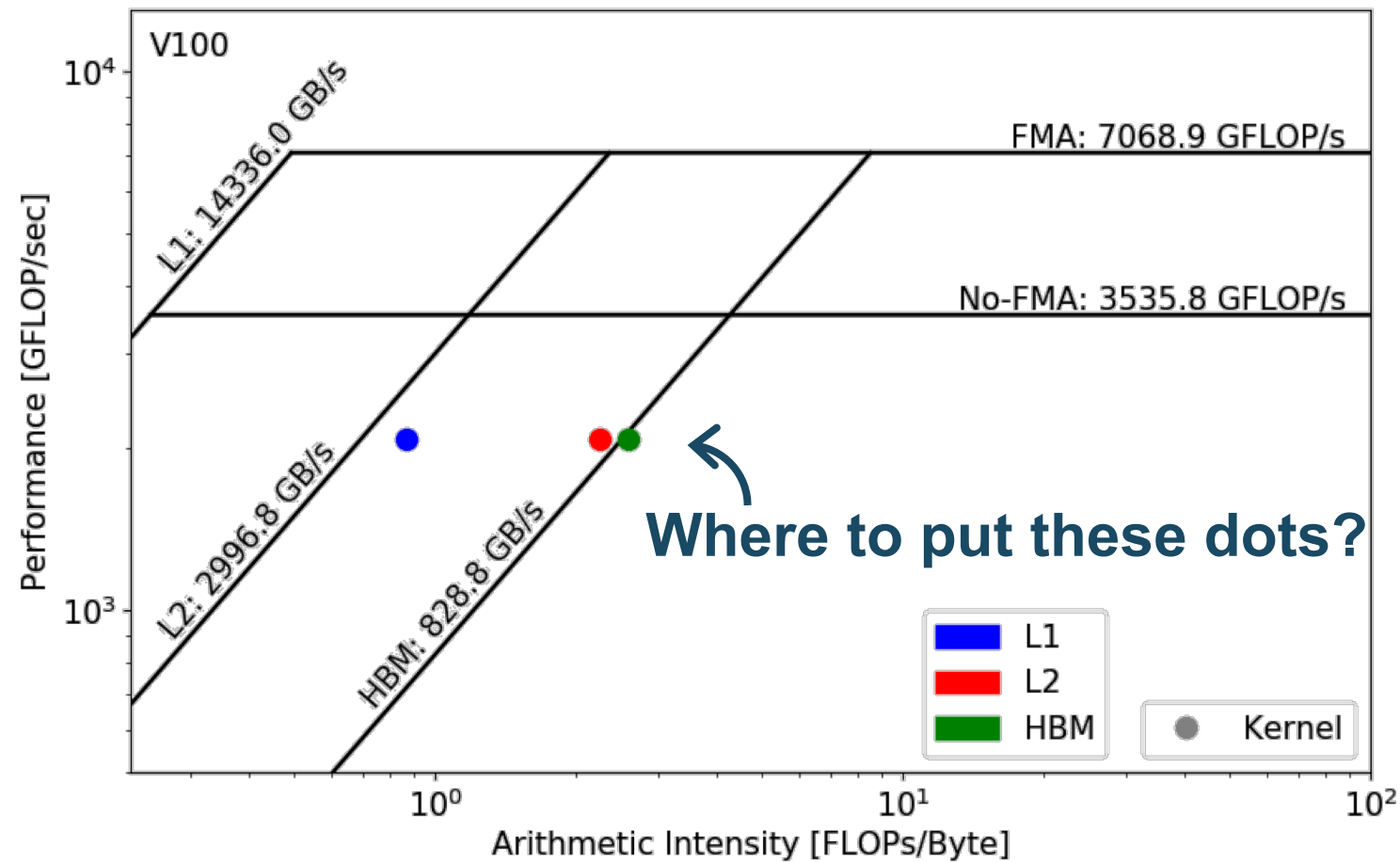
- **You may never achieve 7.8 TFLOP/s**
- **You may be closer to the ceiling than you think you are**



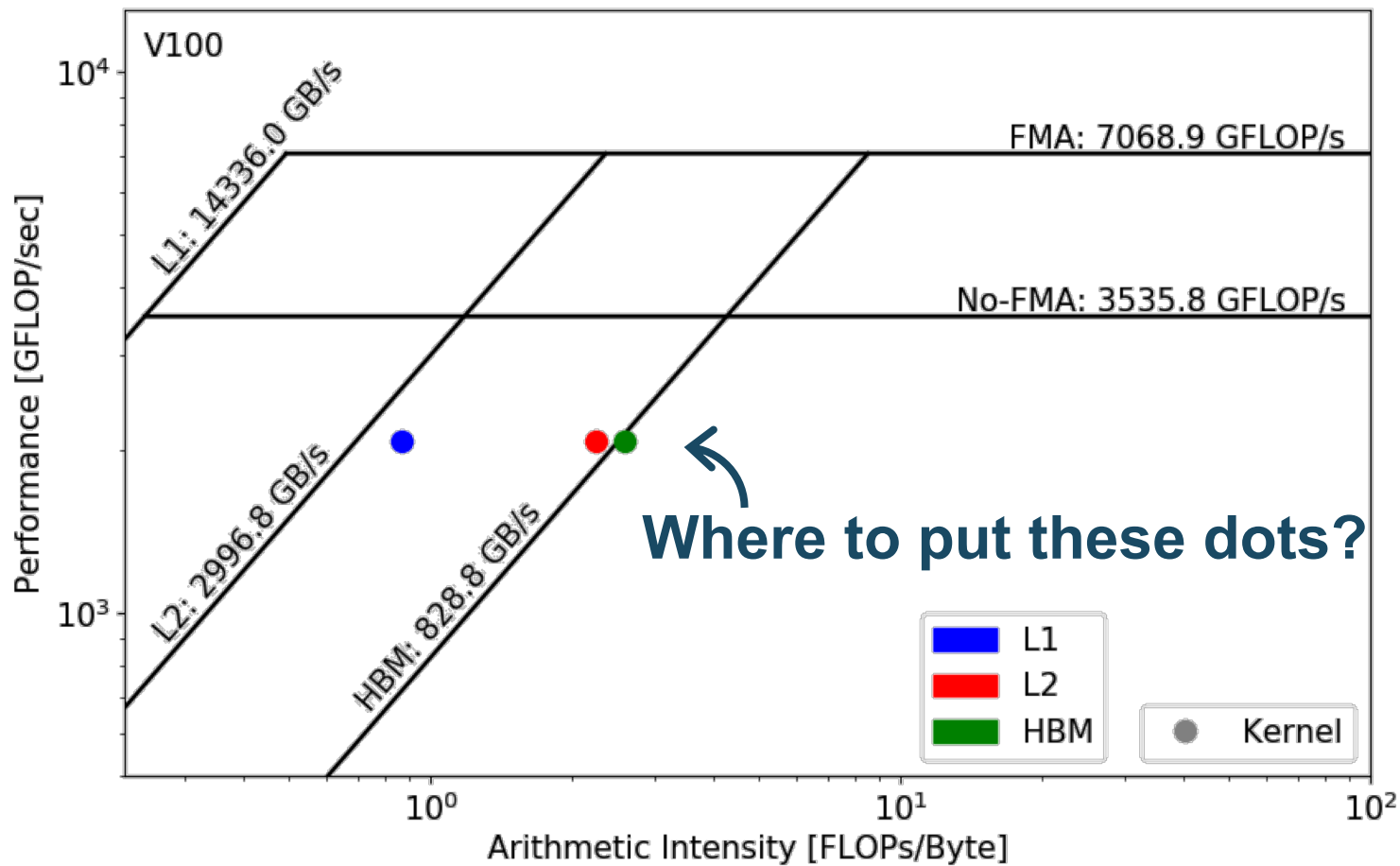
Step 2. Collect Application Performance



Step 2. Collect Application Performance



Step 2. Collect Application Performance



Require three raw measurements:

- Runtime
- FLOPs
- Bytes (on each cache level)

to calculate AI and GFLOP/s:

$$\text{Arithmetic Intensity (FLOPs/Byte)} = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}}$$

$$\text{Performance (GFLOP/s)} = \frac{\text{nvprof FLOPs}}{\text{Runtime}}$$

Collect Application Performance



- Runtime:
 - Time per invocation of a kernel
`nvprof --print-gpu-trace ./application`
 - Average time over multiple invocations
`nvprof --print-gpu-summary ./application`
 - Same kernel with different input parameters are grouped separately
- FLOPs:
 - Predication aware and complex-operation aware (such as divides)
 - `nvprof --kernels 'kernel_name' --metrics 'flop_count_xx' ./application`
 - e.g. `flop_count_{dp/dp_add/dp_mul/dp_fma, sp*, hp*}`

Collect Application Performance



- Bytes for different cache levels in order to construct hierarchical Roofline:
 - Bytes = (read transactions + write transactions) x transaction size
 - `nvprof --kernels 'kernel_name' --metrics 'metric_name'`
`./application`

Level	Metrics	Transaction Size
First Level Cache*	<code>gld_transactions, gst_transactions, atomic_transactions, local_load_transactions, local_store_transactions, shared_load_transactions, shared_store_transactions</code>	32B
Second Level Cache	<code>l2_read_transactions, l2_write_transactions</code>	32B
Device Memory	<code>dram_read_transactions, dram_write_transactions</code>	32B
System Memory	<code>system_read_transactions, system_write_transactions</code>	32B

- Note: surface and texture transactions are ignored here for simplicity (HPC applications)

Example Output

```
[cjyang@voltar source]$ nvprof --kernels "1:7:smooth_kernel:1" --metrics
flop_count_dp --metrics gld_transactions --metrics gst_transactions --
metrics l2_read_transactions --metrics l2_write_transactions --metrics
dram_read_transactions --metrics dram_write_transactions --metrics
system_read_bytes --metrics system_write_bytes ./hpgmg-fv-fp 5 8
```

context : stream : kernel : invocation

- Export to CSV: `--csv -o nvprof.out`

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla V100-PCIE-16GB (0)"					
Kernel: void smooth_kernel<int=6, int=32, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)					
1	flop_count_dp	Floating Point Operations(Double Precision)	30277632	30277632	30277632
1	gld_transactions	Global Load Transactions	4280320	4280320	4280320
1	gst_transactions	Global Store Transactions	73728	73728	73728
1	l2_read_transactions	L2 Read Transactions	890596	890596	890596
1	l2_write_transactions	L2 Write Transactions	85927	85927	85927
1	dram_read_transactions	Device Memory Read Transactions	702911	702911	702911
1	dram_write_transactions	Device Memory Write Transactions	151487	151487	151487
1	system_read_bytes	System Memory Read Bytes	0	0	0
1	system_write_bytes	System Memory Write Bytes	160	160	160

Step 3. Plot Roofline with Python

- Calculate Arithmetic Intensity and GFLOP/s performance
 - x coordinate: Arithmetic Intensity
 - y coordinate: GFLOP/s performance

$$\text{Performance (GFLOP/s)} = \frac{\text{nvprof FLOPs}}{\text{Runtime}}, \quad \text{Arithmetic Intensity (FLOPs/Byte)} = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}}$$

- Plot Roofline with Python Matplotlib
 - Example scripts:
 - <https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting>
 - Tweak as needed for more complex Rooflines

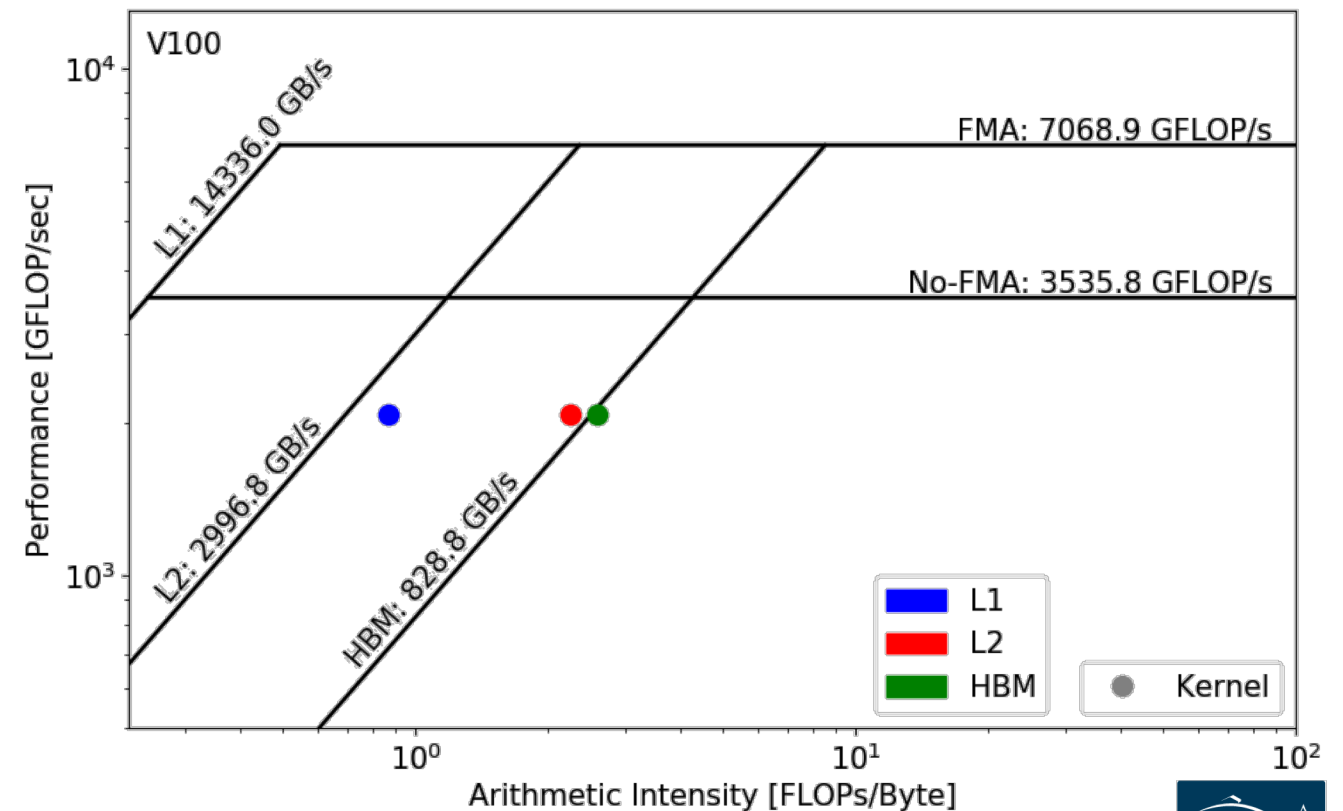
Plot Roofline with Python

- Quick example: `plot_roofline.py data.txt`
- Accepts space-delimited list for values
- Use quotes to separate names/labels

data.txt

```
# all data is space delimited
memroofs 14336.0 2996.8 828.758
mem_roof_names 'L1' 'L2' 'HBM'
comproofs 7068.86 3535.79
comp_roof_names 'FMA' 'No-FMA'

# omit the following if only plotting roofs
# AI: arithmetic intensity; GFLOPs: performance
AI 0.87 2.25 2.58
GFLOPs 2085.756683
labels 'Kernel'
```



Recap: Methodology to Construct Roofline



1. Collect Roofline ceilings

- ERT: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- compute (FMA/no FMA) and bandwidth (DRAM, L2, ...)

2. Collect application performance

- nvprof: `--metrics, --events, --print-gpu-trace`
- FLOPs, bytes (DRAM, L2, ...), runtime

3. Plot Roofline with Python Matplotlib

- arithmetic intensity, GFLOP/s performance, ceilings
- example scripts: <https://github.com/cyanguwa/nersc-roofline>

Recap: Methodology to Construct Roofline



1. Collect Roofline ceilings

- ERT: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- **compute** (FMA/no FMA) and **bandwidth** (DRAM, L2, ...)

2. Collect application performance

- `nvprof: --metrics, --events, --print-gpu-trace`
- FLOPs, bytes (DRAM, L2, ...), runtime

3. Plot Roofline with Python Matplotlib

- arithmetic intensity, GFLOP/s performance, ceilings
- example scripts: <https://github.com/cyanguwa/nersc-roofline>

Recap: Methodology to Construct Roofline



1. Collect Roofline ceilings

- ERT: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- **compute** (FMA/no FMA) and **bandwidth** (DRAM, L2, ...)

2. Collect application performance

- `nvprof: --metrics, --events, --print-gpu-trace`
- **FLOPs**, **bytes** (DRAM, L2, ...), **runtime**

3. Plot Roofline with Python Matplotlib

- arithmetic intensity, GFLOP/s performance, ceilings
- example scripts: <https://github.com/cyanguwa/nersc-roofline>

Recap: Methodology to Construct Roofline



1. Collect Roofline ceilings

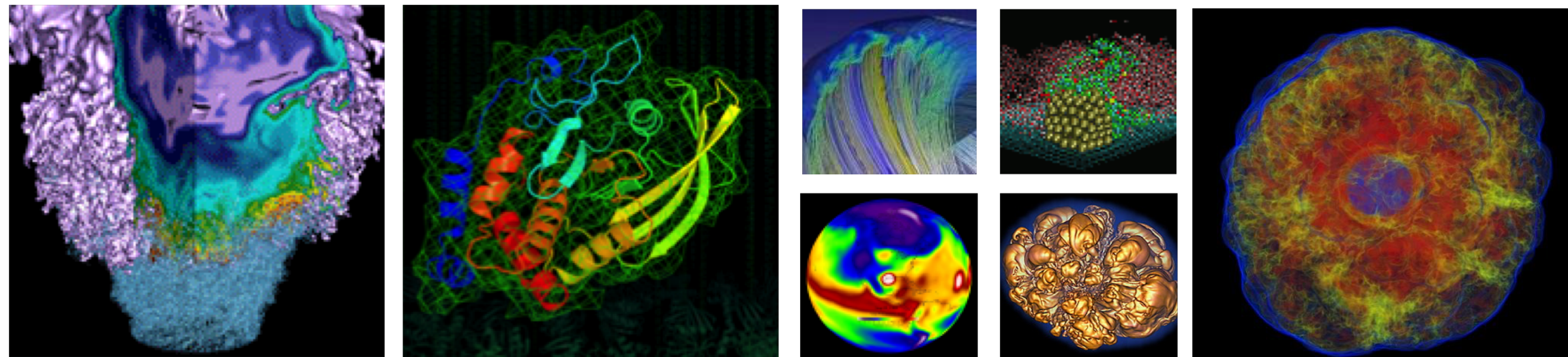
- ERT: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- **compute** (FMA/no FMA) and **bandwidth** (DRAM, L2, ...)

2. Collect application performance

- nvprof: `--metrics, --events, --print-gpu-trace`
- **FLOPs, bytes** (DRAM, L2, ...), **runtime**

3. Plot Roofline with Python Matplotlib

- **arithmetic intensity, GFLOP/s** performance, **ceilings**
- example scripts: <https://github.com/cyanguwa/nersc-roofline>



Roofline Analysis with Use Cases



U.S. DEPARTMENT OF
ENERGY

Office of
Science



BERKELEY LAB
Lawrence Berkeley National Laboratory

Code Example 1: GPP



- GPP (General Plasmon Pole) kernel from BerkeleyGW (Material Science)
- <https://github.com/cyanguwa/BerkeleyGW-GPP>
- Medium problem size: 512 2 32768 20
- Tensor-contraction, abundant parallelism, large reductions
- Low FMA counts, divides, complex double data type, HBM data 1.5GB

Pseudo Code

```
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do ig = 1, ncouls         #threadIdx.x
      do iw = 1, nw           #unrolled
        compute; reductions
```

Code Example 1: GPP



- Three experiments:

Vary <code>nw</code> from 1 to 6	To study impact of varying Arithmetic Intensity on performance
Compile w/wo FMA	To study impact of instruction mix on performance on performance
Stride <code>ig</code> loop	To study impact of suboptimal memory coalescing on performance

- Note that `nvprof` has already taken care of
 - Appropriate counting of FLOPs for complex instructions
 - `div`, `exp`, `log` and `sin/cos` should be counted as multiple FLOPs rather than 1
 - Appropriate counting of FLOPs for predicated-out threads
 - FLOPs are only counted on non-predicated threads

Code Example 1: GPP



- Highly parameterizable
 1. Varying `nw` from 1 to 6 to increase arithmetic intensity
 - FLOPs increases, but data movement stays (at least for HBM)

Pseudo Code

```
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do ig = 1, ncouls         #threadsIdx.x
      do iw = 1, nw         #unrolled
        compute; reductions
```

2. Compiling with and without FMA
 - `-fmad=true/false`

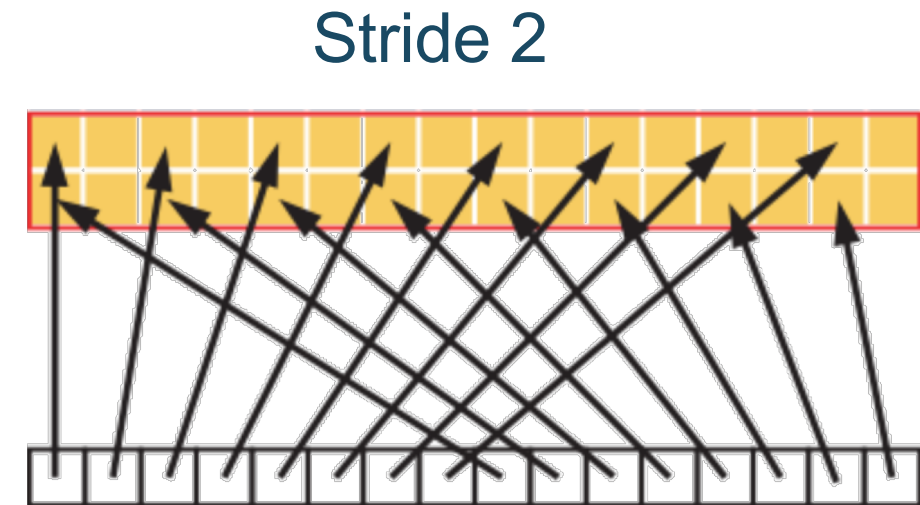
Code Example 1: GPP

- Highly parameterizable
 3. Striding `ig` loop to analyze impact of suboptimal memory coalescing
 - Split `ig` loop to two loops and place the ‘blocking’ loop outside

Pseudo Code

```

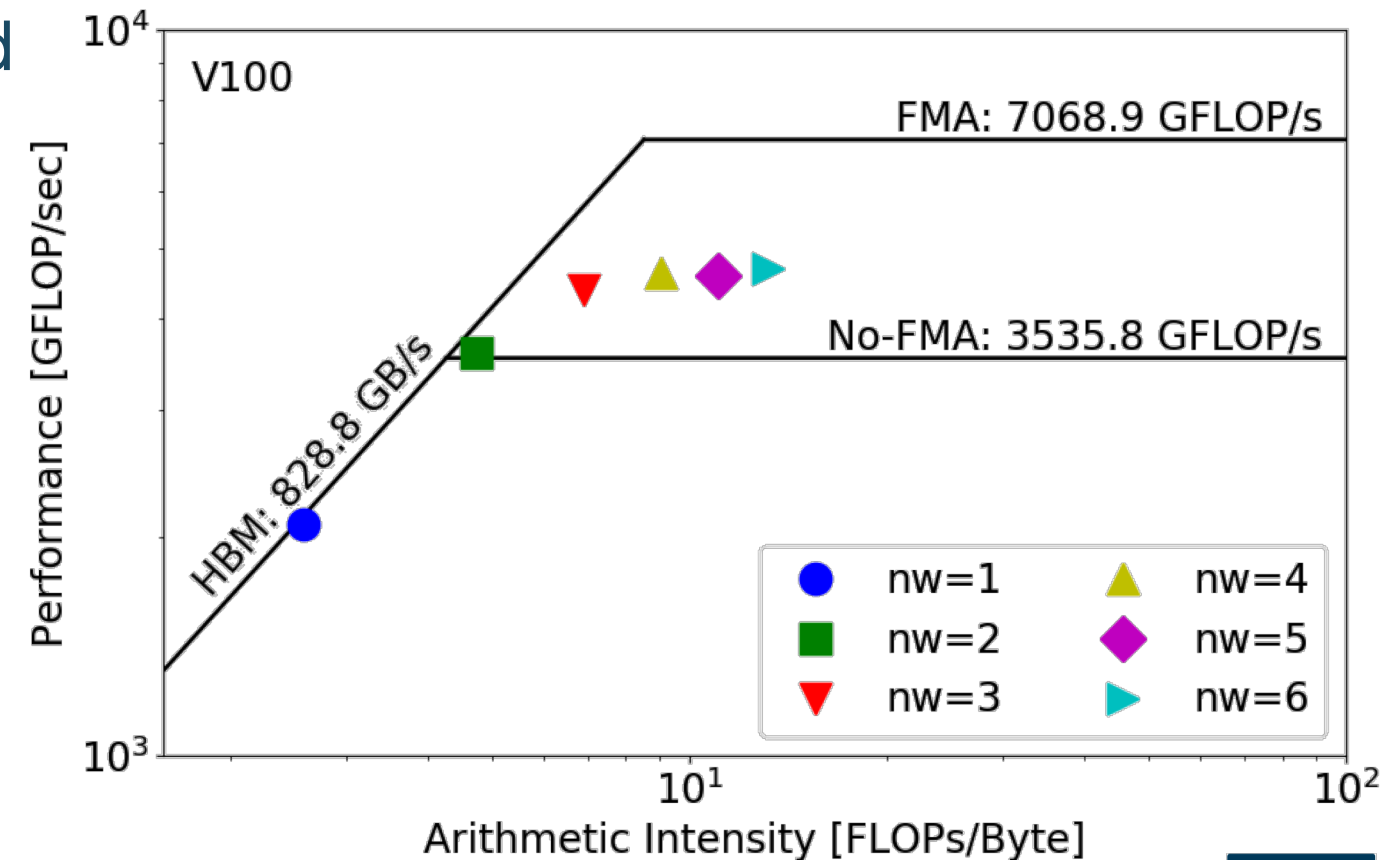
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do igs = 0, stride - 1
      do ig = 1, ncouls/stride #threadIdx.x
        do iw = 1, nw         #unrolled
          compute; reductions
        
```



Code Example 1: GPP



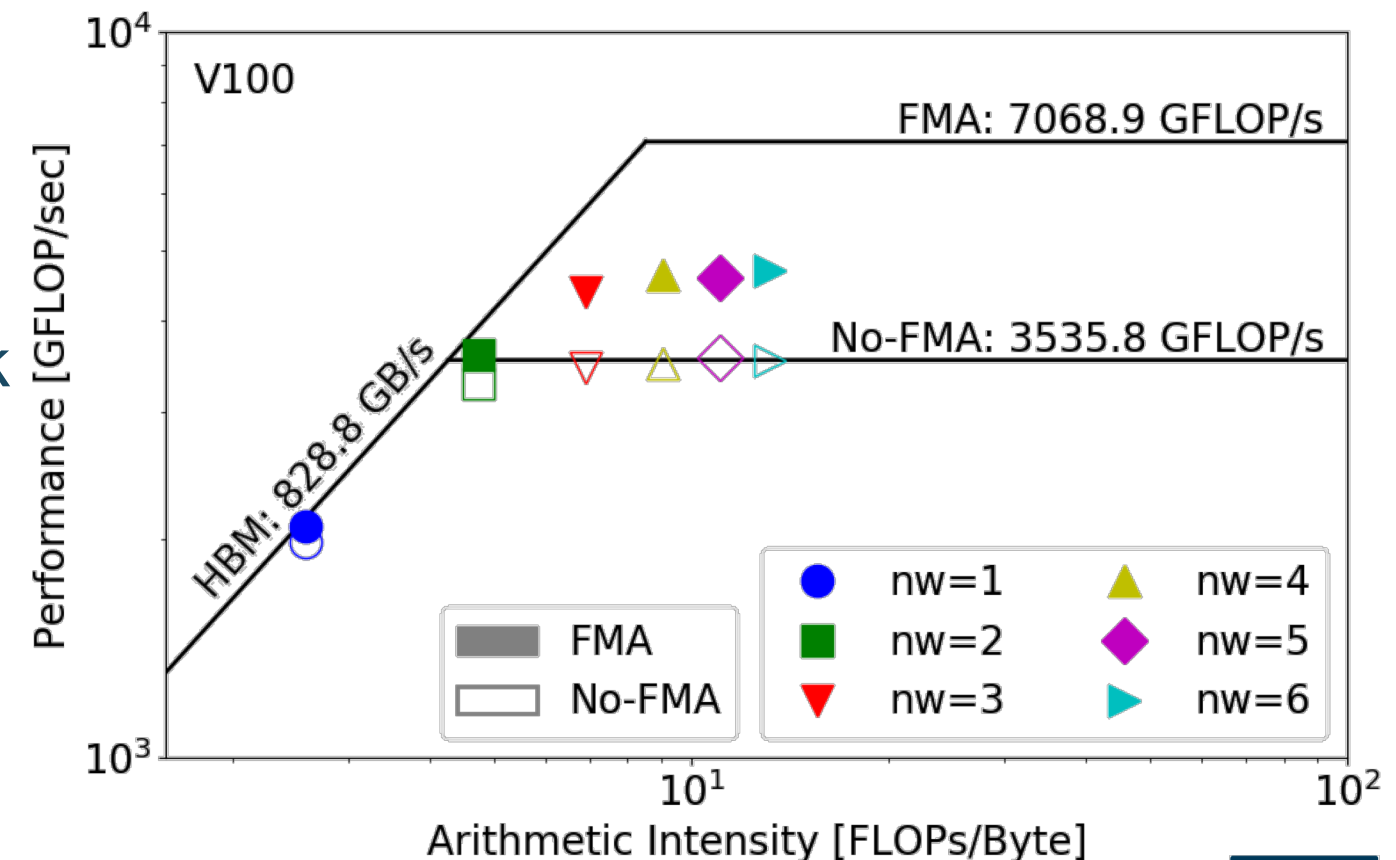
- **Experiments 1:** study the impact of varying AI on performance
- HBM Roofline, i.e. bytes are HBM bytes
 - AI increases as nw grows
 - GPP moves from a bandwidth bound region to a compute bound region
- Roofline captures the change in AI



Code Example 1: GPP



- **Experiments 1 & 2:** study the impact of instruction mix on performance
- HBM Roofline, i.e. bytes are HBM bytes
 - No-FMA performance converges to the no-FMA ceiling, but FMA performance is still far from the FMA ceiling
 - Not reaching FMA ceiling due to lack of FMA instructions
- Roofline captures effects of instruction mix



Code Example 1: GPP



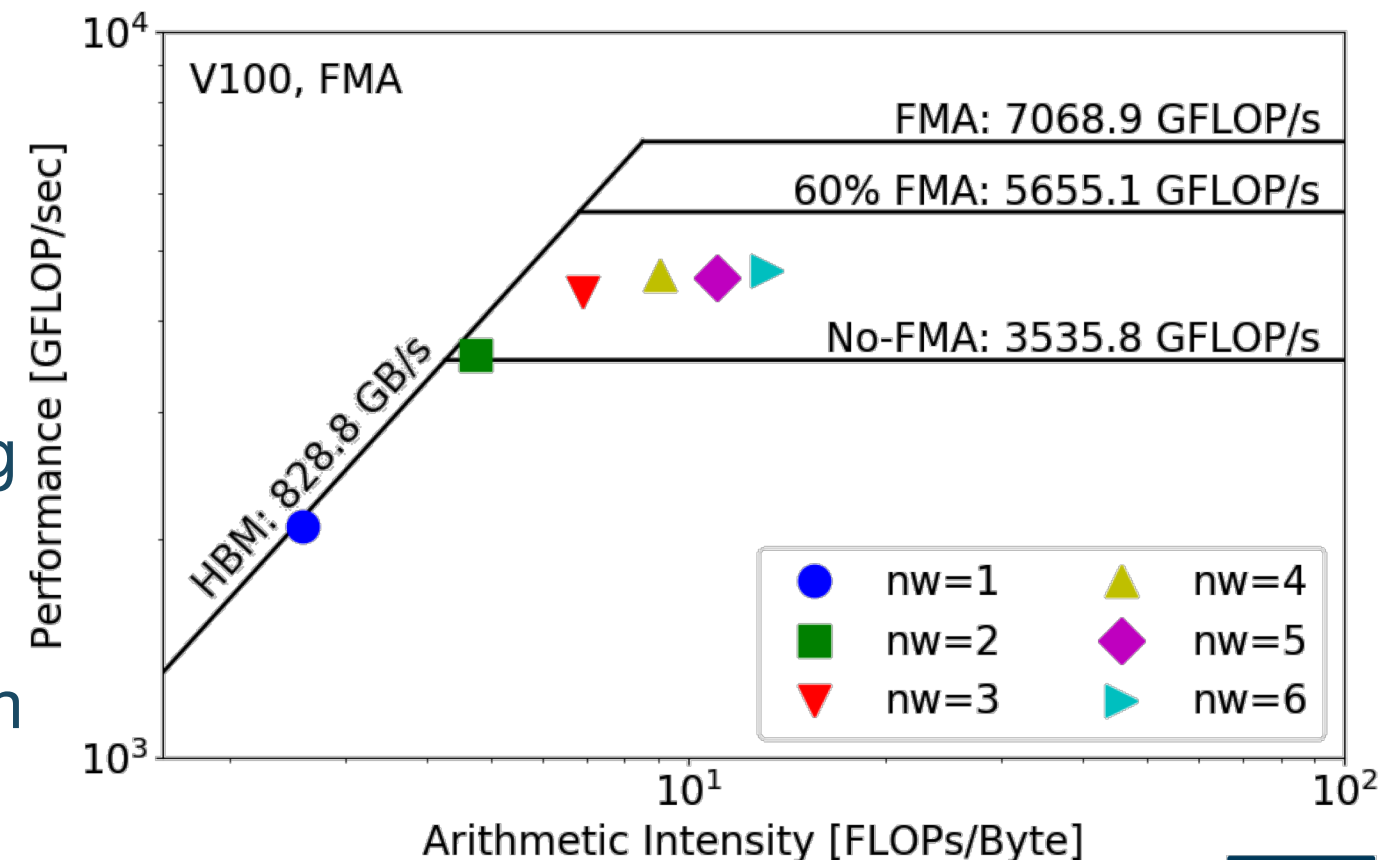
- **Experiments 1 & 2:** study the impact of instruction mix on performance
- At $nw=6$, GPP has $\alpha = \frac{\text{FMA FP64 instr.}}{\text{FMA FP64 instr.} + \text{non-FMA FP64 instr.}} = 60\%$ of FMA instructions

- Expected performance is

$$\beta = \frac{\alpha \times 2 + (1 - \alpha)}{2} = 80\% \text{ of compute peak.}$$

But at $nw=6$, GPP is only achieving **66%**.

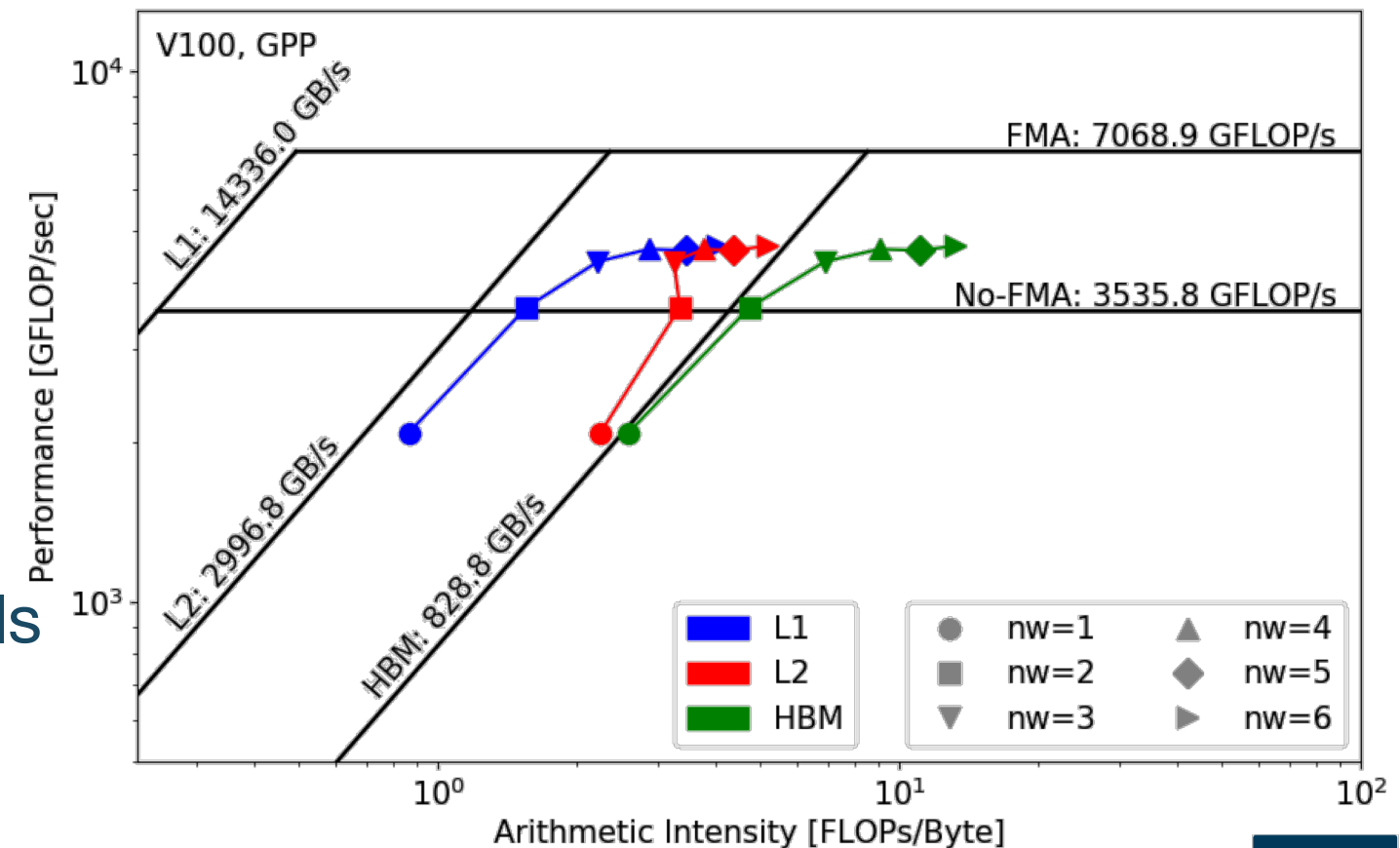
- Other FP/non-FP instructions may be taking up the instruction issue/execution pipeline
- Partial Roofline can show you the headroom



Code Example 1: GPP



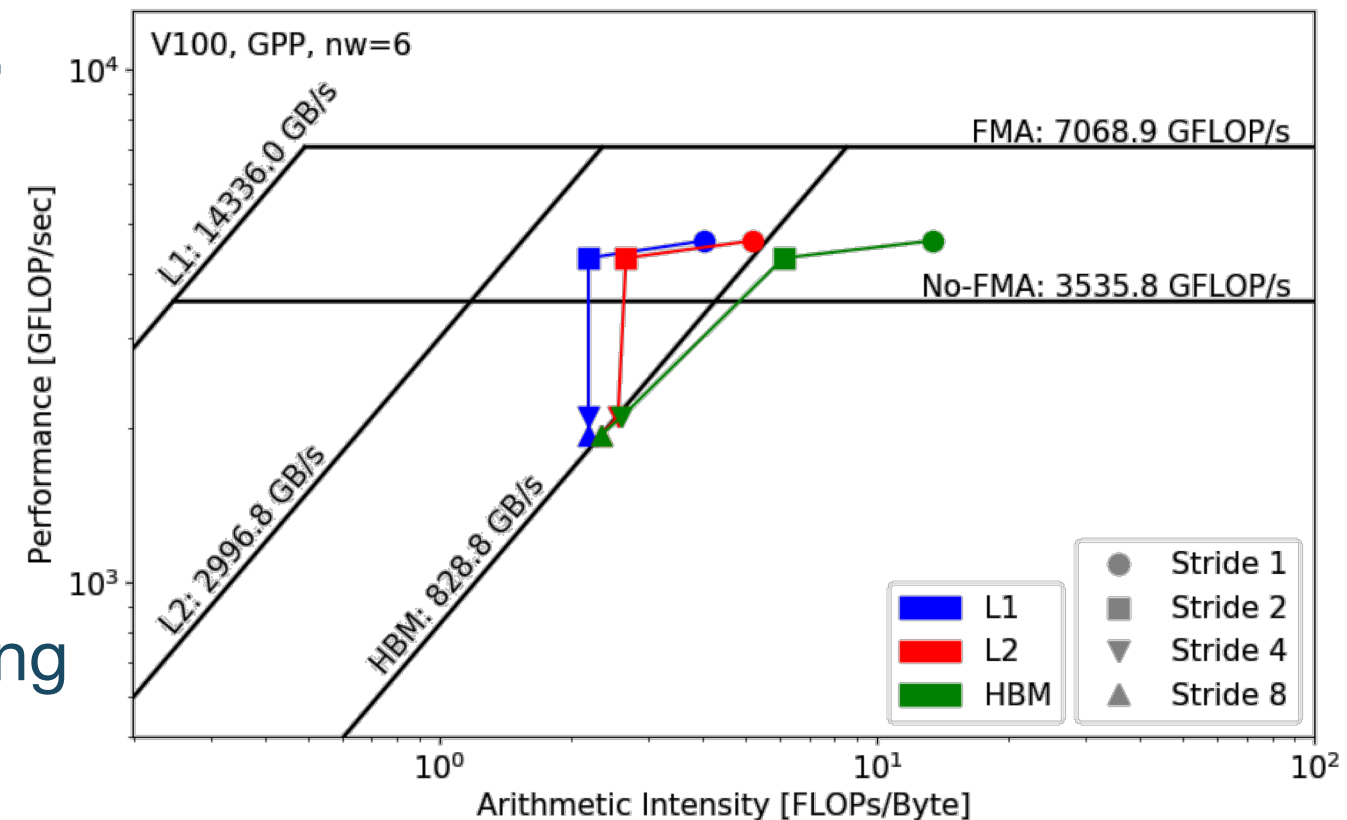
- **Experiments 1 & 2:** What else is going on?
- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
 - GPP is HBM bound at low nw 's and compute bound at high nw 's
 - FLOPs $\propto nw$
 - HBM bytes: constant
 - L2 bytes: increasing at $\alpha > 1$
 - L1 bytes: constant
 - Spike in L2 curve at $nw=2, 3$
- Hierarchical Roofline captures more details about cache locality



Code Example 1: GPP

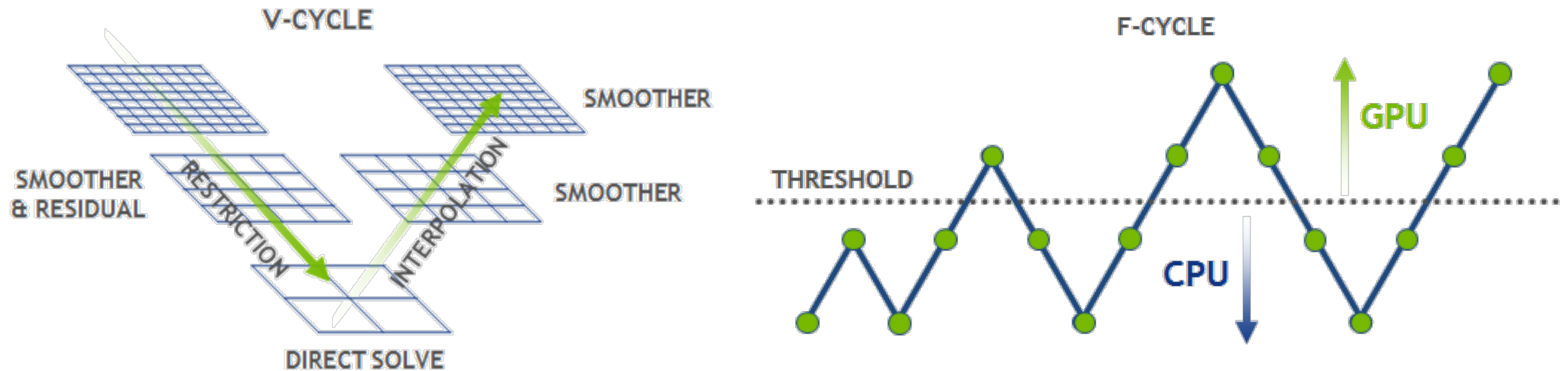


- **Experiment 3:** study the effects of suboptimal memory coalescing
 - $nw=6$
- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
 - L1/L2 bytes doubles from stride 1 to 2, but stays almost constant afterwards
 - at $nw=6$, GPP moves from compute bound to bandwidth bound
 - Eventually all dots converge to HBM
- Roofline captures effects of memory coalescing



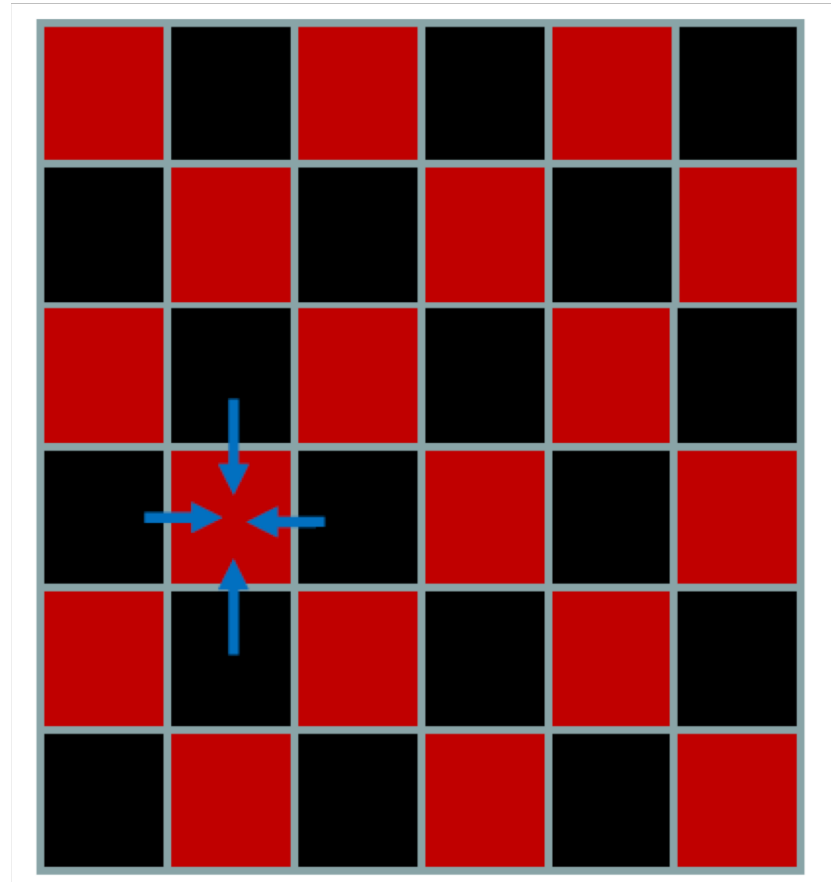
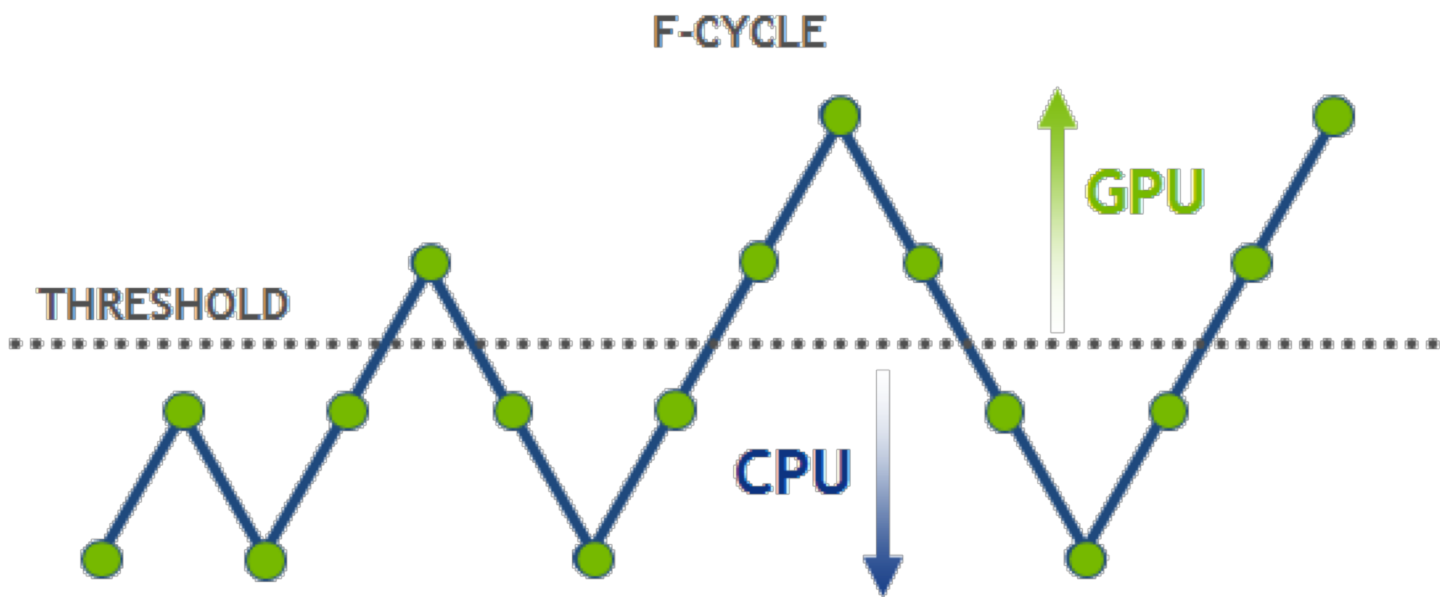
Code Example 2: HPGMG

- HPGMG (High-performance Geometric Multigrid) from Adaptive Mesh Refinement codes
- <https://bitbucket.org/nsakharnykh/hpgmg-cuda>
- Stencil code, F-cycles and V-cycles, GSRB smoother kernel (Gauss-Seidel Red-Black)



Code Example 2: HPGMG

- Hybrid GPU and CPU code
 - Example: `hpgmg-fv 7 8`
 - 128^3 box x 8, Level 5-8 run on GPU, Level 1-4 on CPU
- Three versions of GSRB kernel
 - GSRB_FP, GSRB_BRANCH, GSRB_STRIDE2



Code Example 2: HPGMG



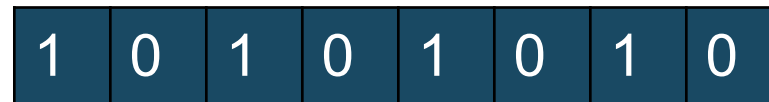
GSRB_FP

```
for(int k=klo; k<(klo+kdim); k++){
  const int ijk = i + j*jStride + k*kStride;
  const double *__restrict__ RedBlack =
    level.RedBlack_FP + ghosts*(1+jStride)
    + ((k^color000)&1)*kStride;
  const double Ax = apply_op_ijk();
  const double lambda = Dinv_ijk();
  const int ij = i + j*jStride;
  xo[ijk] = X(ijk) + RedBlack[ij]*lambda*(rhs[ijk]-Ax);
}
```



8 elements

Sweep



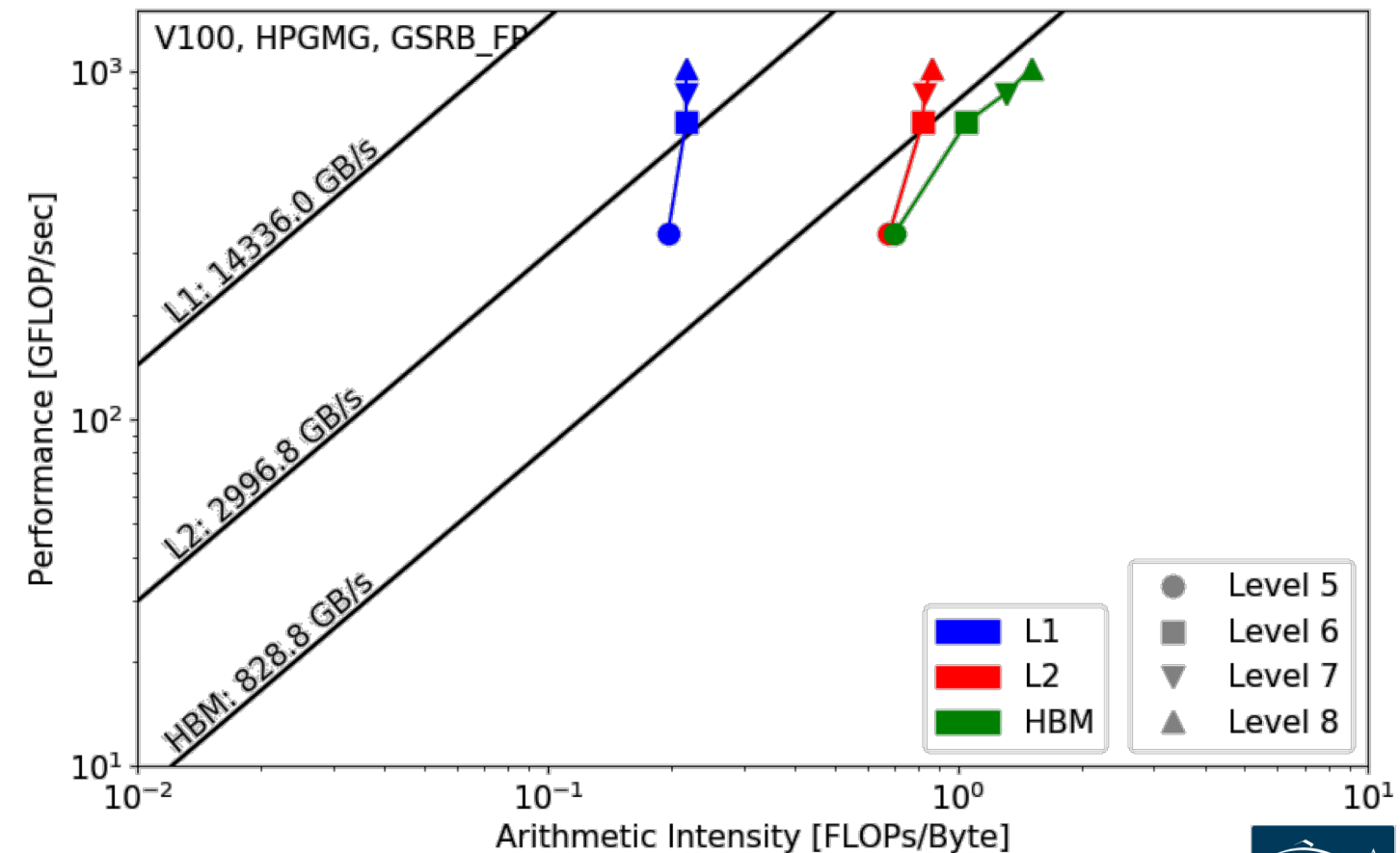
8 threads

Code Example 2: HPGMG



GSRB_FP

- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
- Highly bandwidth bound, inherent to stencil codes
- From Level 5 to Level 8:
 - AI slightly increases due to better Surface: Volume ratio
 - More HBM bound as more data is read in
- Roofline captures computational characteristics of the algorithm



Code Example 2: HPGMG



GSRB_FP

```
for(int k=klo; k<(klo+kdim); k++){
  const int ijk = i + j*jStride + k*kStride;
  const double *__restrict__ RedBlack =
    level.RedBlack_FP + ghosts*(1+jStride)
    + ((k^color000)&1)*kStride;
  const double Ax = apply_op_ijk();
  const double lambda = Dinv_ijk();
  const int ij = i + j*jStride;
  xo[ijk] = X(ijk) + RedBlack[ij]*lambda*(rhs[ijk]-Ax);
}
```

GSRB_BRANCH

```
for(int k=klo; k<klo+kdim; k++){
  const int ijk = i + j*jStride + k*kStride;
  if(((i^j^k^color000^1)&1)){
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
  }else{
    xo[ijk] = X(ijk);
  }
}
```

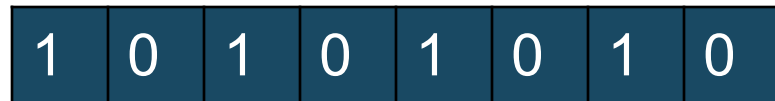


8 elements



8 elements

Sweep



8 threads



8 threads

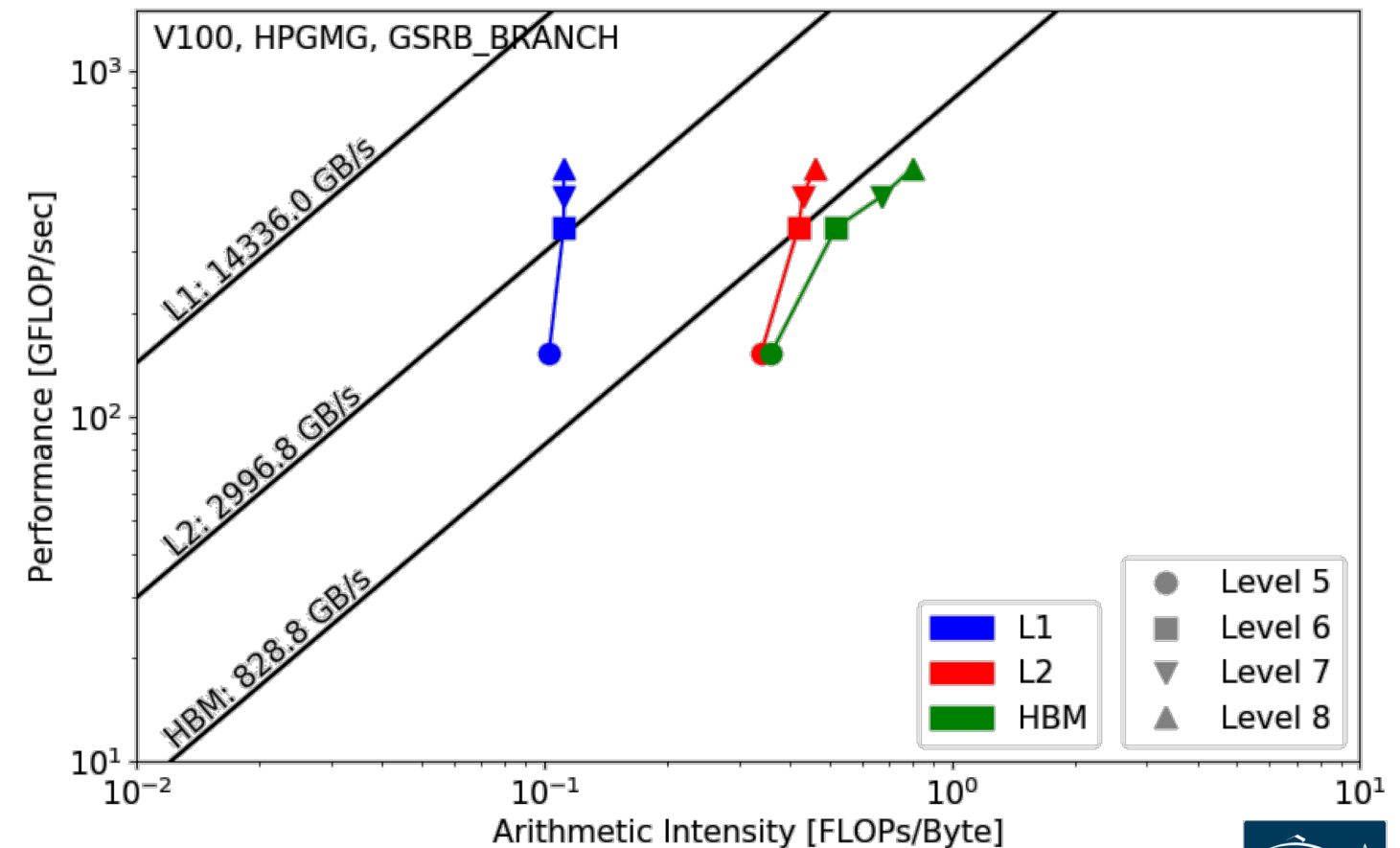
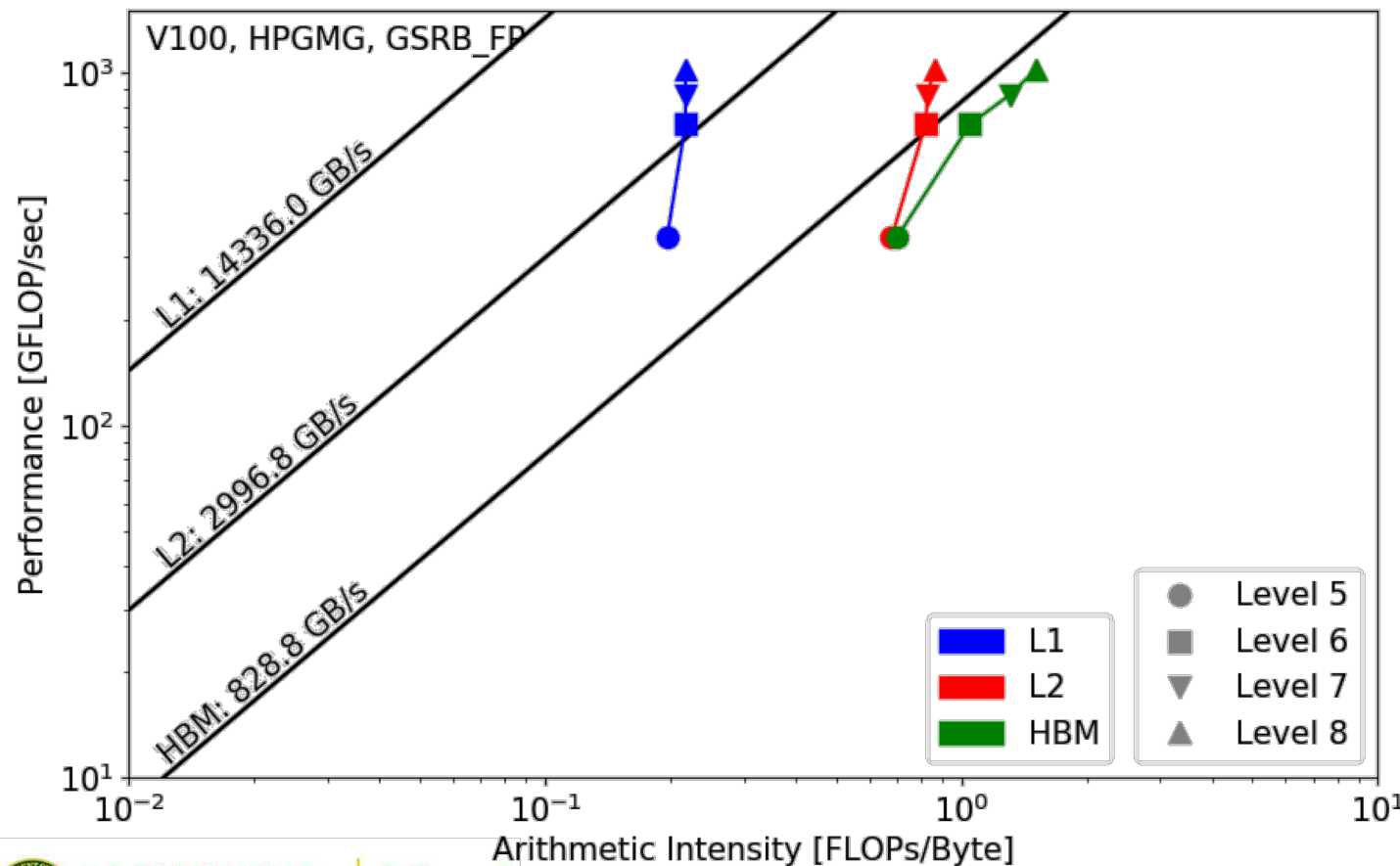
- GSRB_BRANCH has half the FLOPs as GSRB_FP but the same HBM/L1/L2 bytes

Code Example 2: HPGMG



GSRB_FP vs. GSRB_BRANCH

- FLOPs halves, bytes doesn't change, thus AI halves and GFLOP/s halves
- Runtime is comparable even though GFLOP/s has halved
- Same number of threads occupied, only with half predicated in GSRB_BRANCH

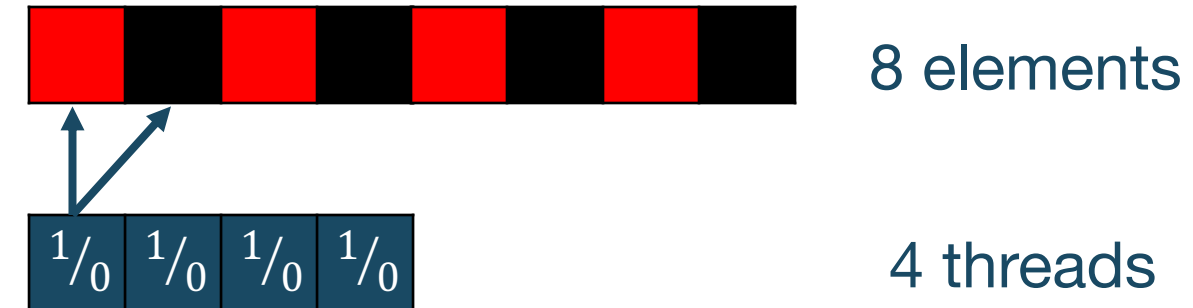


Code Example 2: HPGMG



GSRB_STRIDE2

```
for(int k=klo; k<klo+kdim; k++){
  i = ilo + !((ilo^j^k^color000)&1) + threadIdx.x*2;
  if(i < ilo+idim){
    const int ijk = i + j*jStride + k*kStride;
    xo[ijk] = X(ijk);
  }
  i = ilo + ((ilo^j^k^color000)&1) + threadIdx.x*2;
  if(i < ilo+idim){
    const int ijk = i + j*jStride + k*kStride;
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
  }
}
```



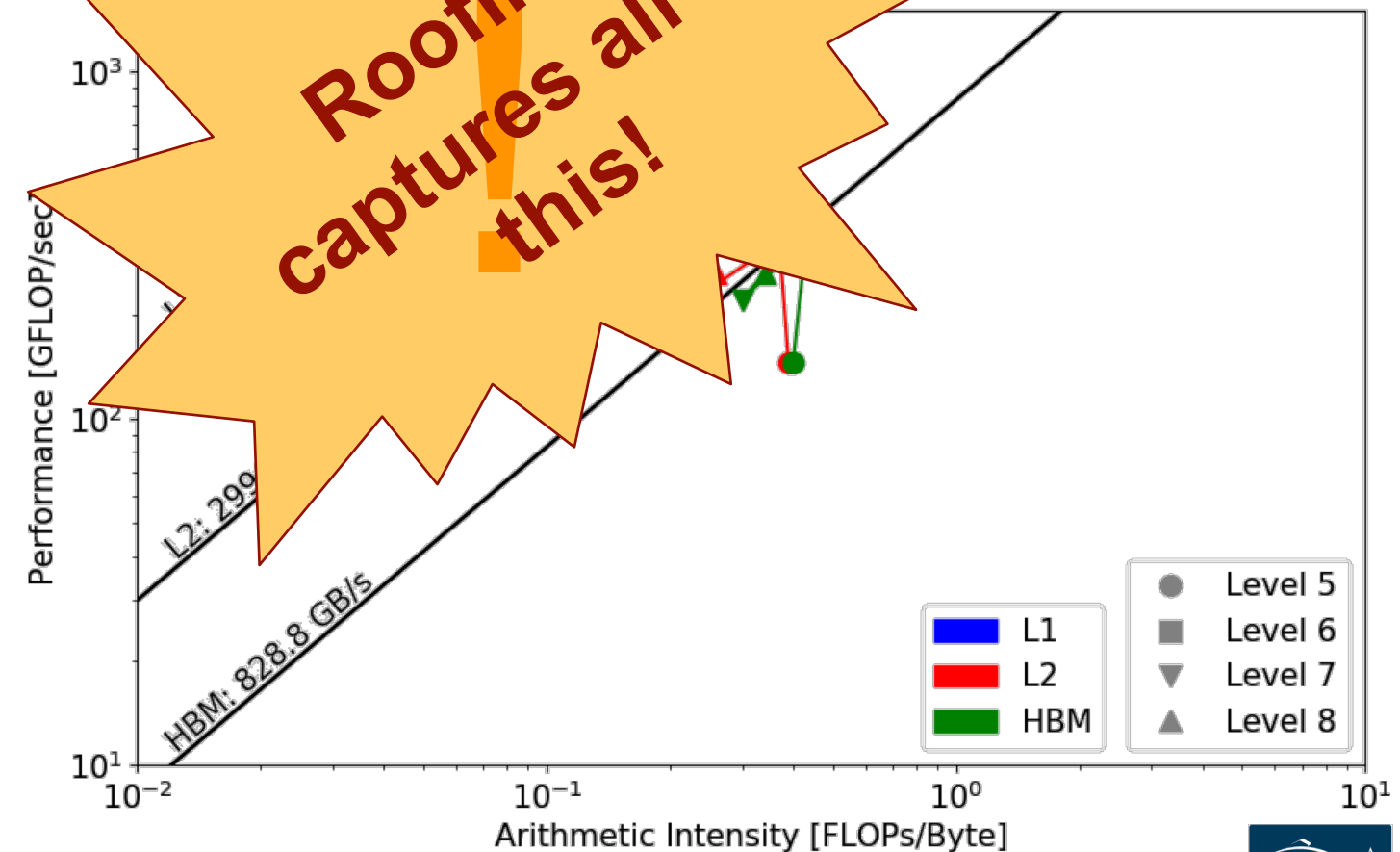
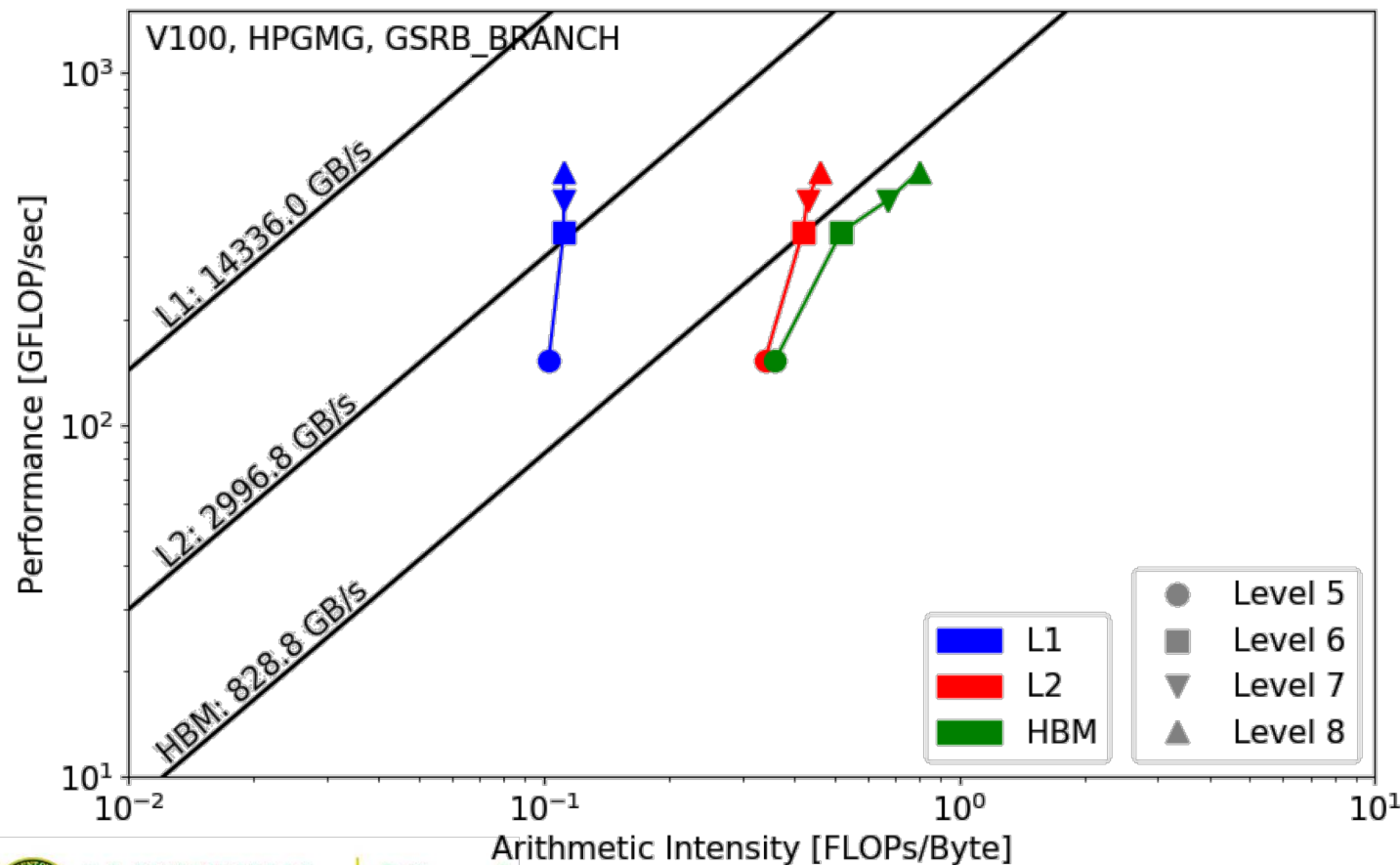
- GSRB_STRIDE2 should have the same FLOPs as GSRB_BRANCH, but same bytes? More writes than GSRB_BRANCH?

Code Example 2: HPGMG

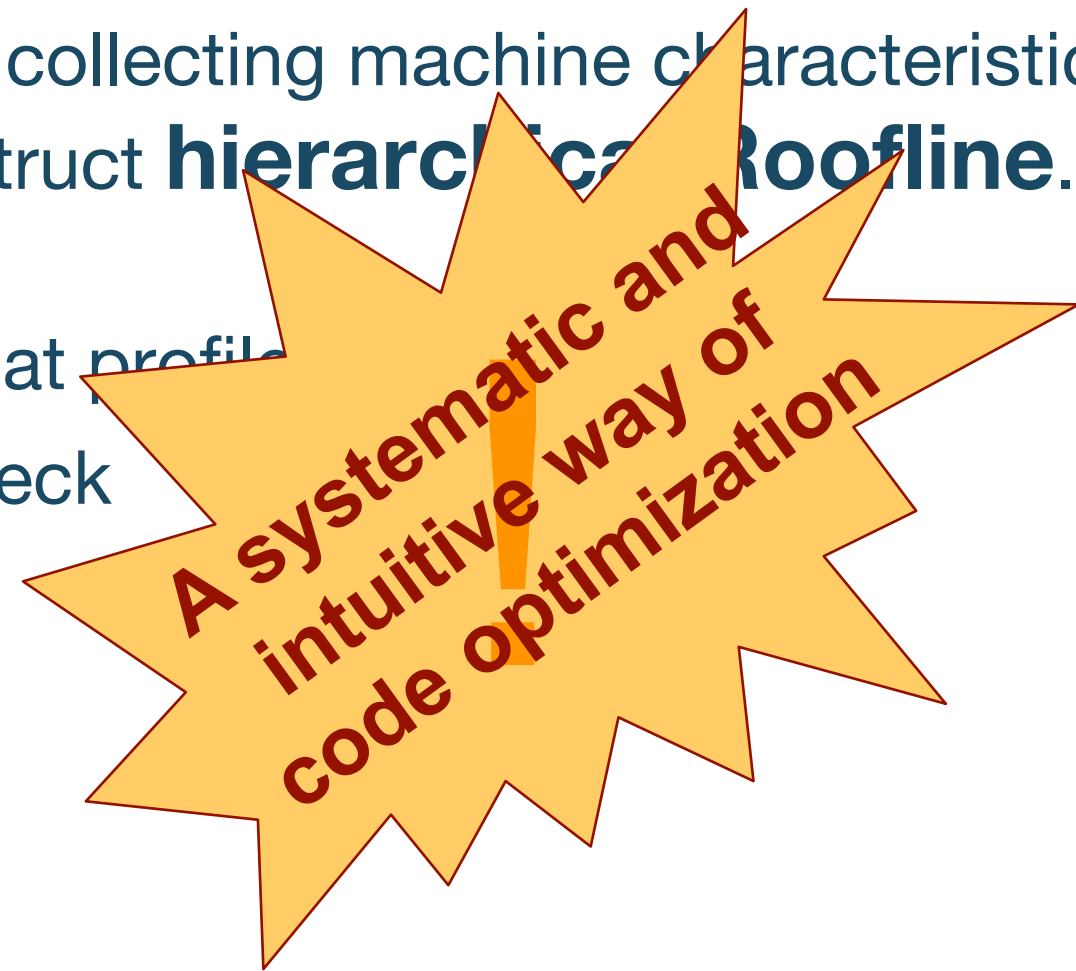


GSRB_BRANCH vs. GSRB_STRIDE2

- Extra writes in GSRB_STRIDE2 cause more capacity misses in L2, leading to AI drop on L2 and DRAM, starting from Level 7 (data size \approx L2 cache size)
- Runtime almost doubled and GFLOP/s halved



- Roofline can gracefully capture various aspects of application performance and architecture characteristics such as arithmetic intensity, instruction mix, memory coalescing and thread predication.
- The proposed methodology is effective in collecting machine characteristics and application data on NVIDIA GPUs to construct **hierarchical Roofline**.
- The Roofline model provides **insights** that profile
 - identify the most immediate bottleneck
 - prioritize optimization efforts
 - tell you when you can stop



Reference



- S. Williams, A. Waterman and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009
- Empirical Roofline Toolkit (ERT): <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- Example scripts for plotting Roofline: <https://github.com/cyanguwa/nersc-roofline>
- General Plasmon Pole kernel: <https://github.com/cyanguwa/BerkeleyGW-GPP>
- HPGMG-CUDA kernel: <https://bitbucket.org/nsakharnykh/hpgmg-cuda>

Acknowledgement



- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This material is based upon work supported by the DOE RAPIDS SciDAC Institute.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02- 05CH11231.



U.S. DEPARTMENT OF
ENERGY

Office of
Science



Thank You!

