

Introduction to the Roofline Model

Samuel Williams

Computational Research Division
Lawrence Berkeley National Lab

SWWilliams@lbl.gov

Acknowledgements

- 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.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Why Use Performance Models or Tools?

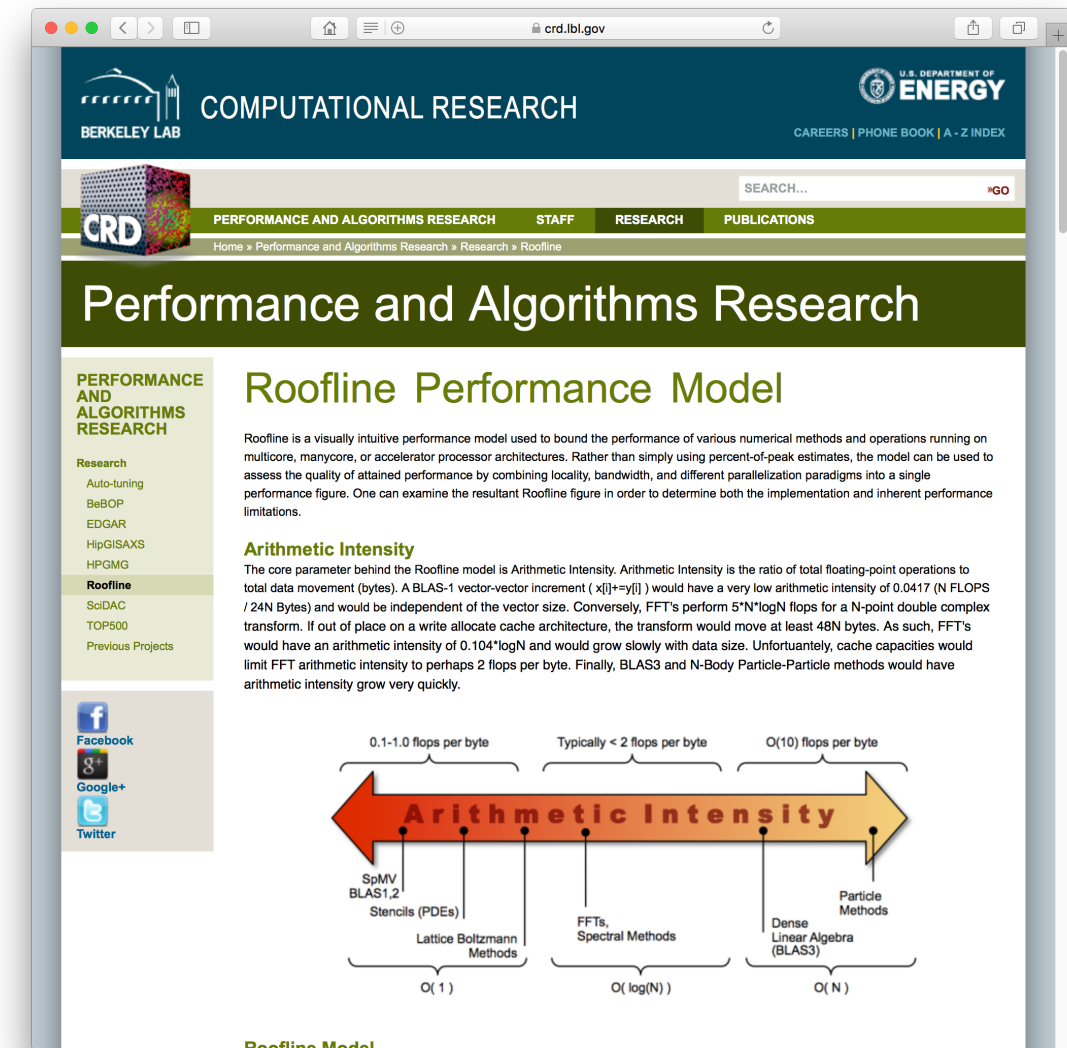
- Identify performance bottlenecks
- Motivate software optimizations
- **Determine when we're done optimizing**
 - Assess performance relative to machine capabilities
 - Motivate need for algorithmic changes
- Predict performance on future machines / architectures
 - Sets realistic expectations on performance for future procurements
 - Used for HW/SW Co-Design to ensure future architectures are well-suited for the computational needs of today's applications.

Performance Models / Simulators

- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)
- The last two decades saw a number of latency-hiding techniques...
 - Out-of-order execution (hardware discovers parallelism to hide latency)
 - HW stream prefetching (hardware speculatively loads data)
 - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)
- Effectively latency hiding has resulted in a shift from a latency-limited computing regime to a **throughput-limited computing regime**

Roofline Model

- The **Roofline Model** is a throughput-oriented performance model...
 - Tracks rates not times
 - Augmented with Little's Law
(concurrency = latency*bandwidth)
 - Independent of ISA and architecture
(applies to CPUs, GPUs, Google TPUs¹, etc...)
- Three Components:
 - Machine Characterization
(realistic performance potential of the system)
 - Application Execution Monitoring
 - Theoretical Application Bounds
(how well could my app perform with perfect compilers, caches, overlap, ...)

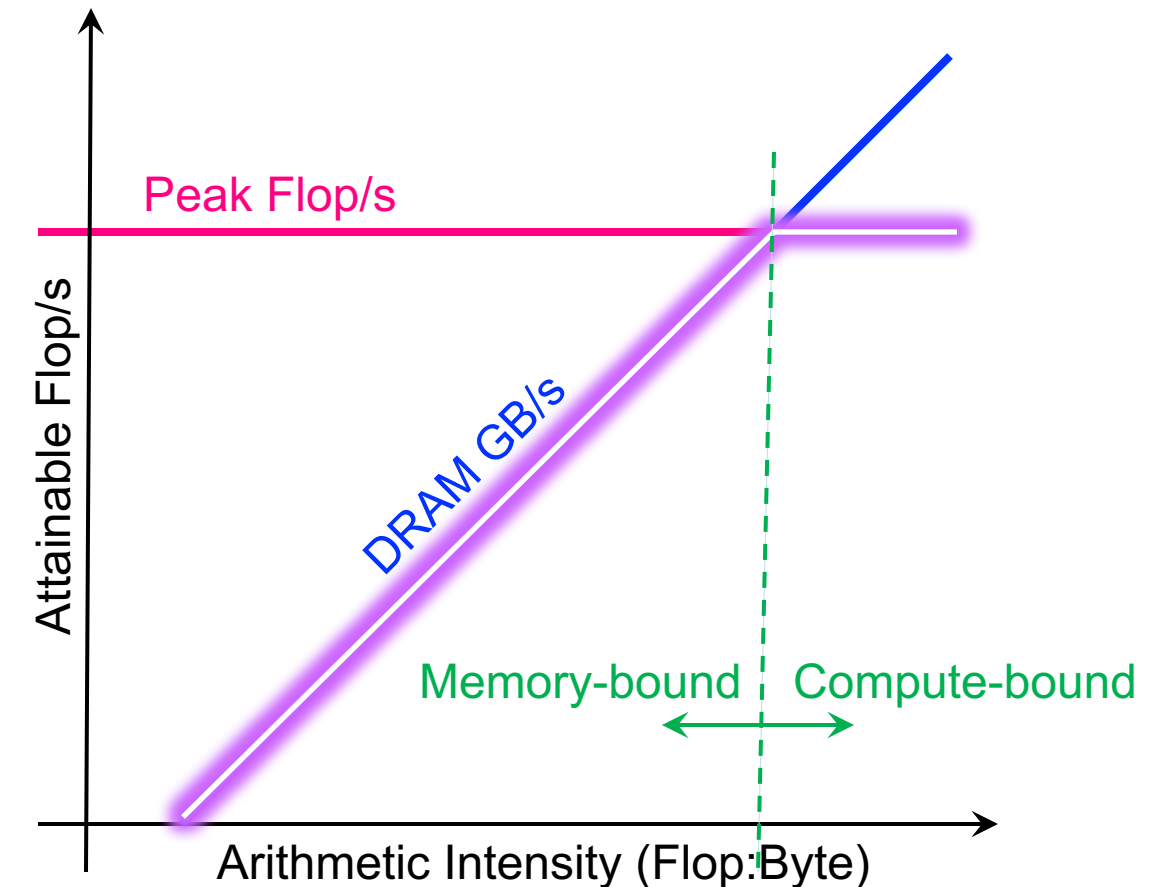


<https://crd.lbl.gov/departments/computer-science/PAR/research/roofline>

¹Jouppi et al, "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA, 2017.

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Consider idealized processor/caches
- Plot the performance bound using Arithmetic Intensity (AI) as the x-axis...
 - $AI = \text{Flops} / \text{Bytes presented to DRAM}$
 - **Attainable Flop/s = min(peak Flop/s, AI * peak GB/s)**
 - **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...)



Roofline Example #1

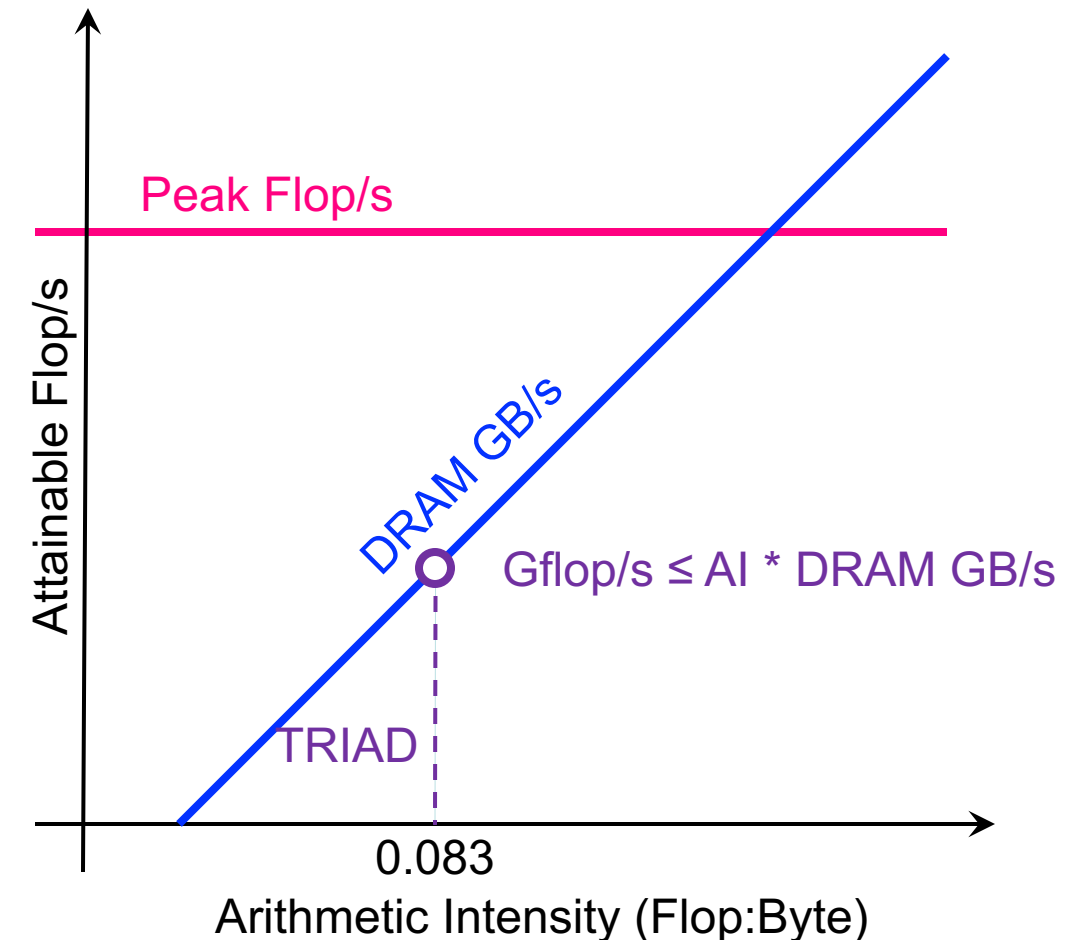
- Typical machine balance is 5-10 flops per byte...

- 40-80 flops per double to exploit compute capability
- Artifact of technology and money
- **Unlikely to improve**

- Consider STREAM Triad...

```
#pragma omp parallel for
for(i=0;i<N;i++){
    Z[i] = X[i] + alpha*Y[i];
}
```

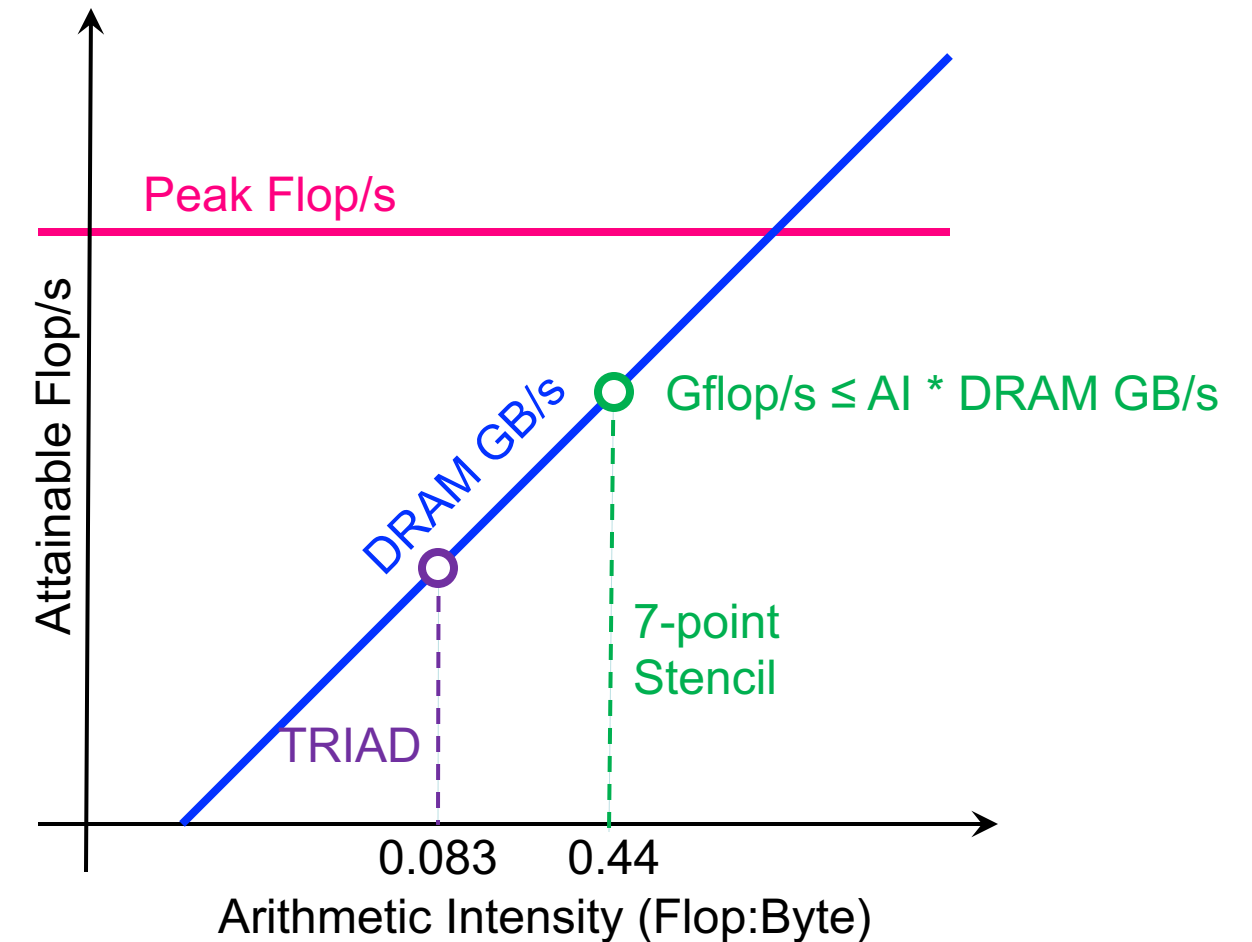
- 2 flops per iteration
- Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
- **AI = 0.083 flops per byte == Memory bound**



Roofline Example #2

- Conversely, 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, but 5x the flop rate**

```
#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++){
    int ijk = i + j*jStride + k*kStride;
    new[ijk] = -6.0*old[ijk
        + old[ijk-1
        + old[ijk+1
        + old[ijk-jStride]
        + old[ijk+jStride]
        + old[ijk-kStride]
        + old[ijk+kStride];
    }}}}
```

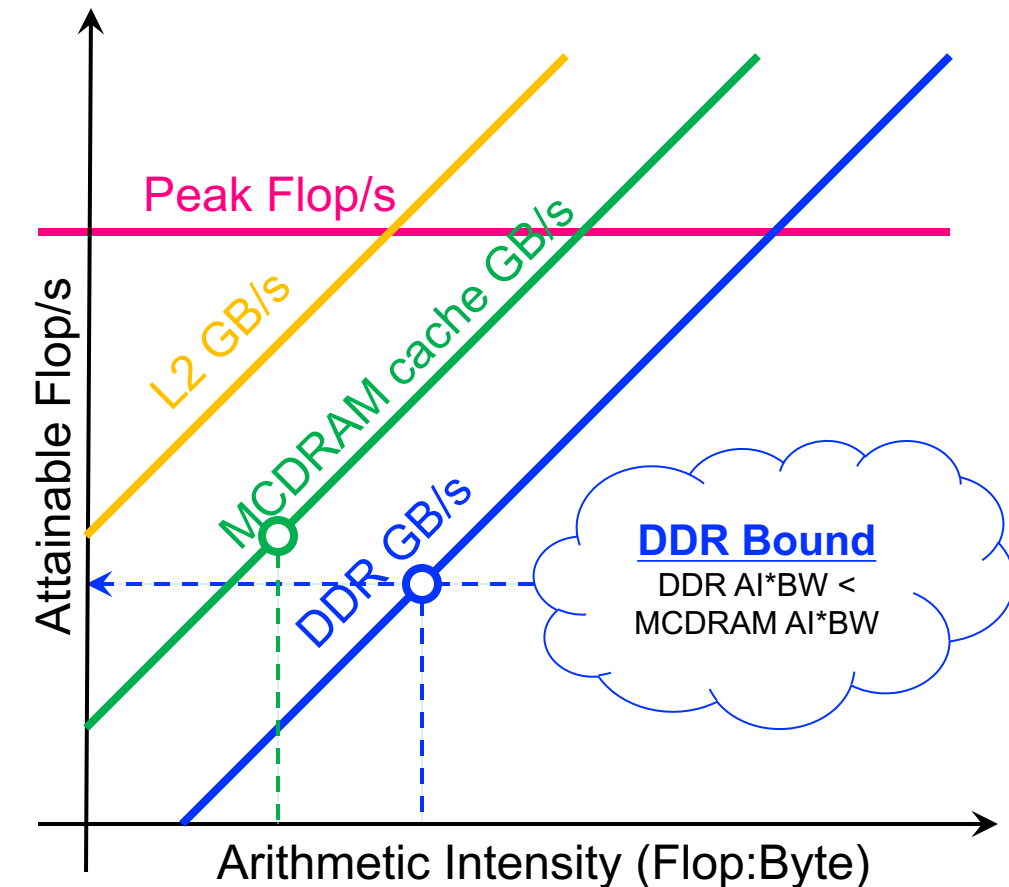


Hierarchical Roofline

- Real processors have multiple levels of memory
 - Registers
 - L1, L2, L3 cache
 - MCDRAM/HBM (KNL/GPU device memory)
 - DDR (main memory)
 - NVRAM (non-volatile memory)
- Applications can have locality in each level
 - Unique data movements imply unique AI's
 - Moreover, each level will have a unique bandwidth

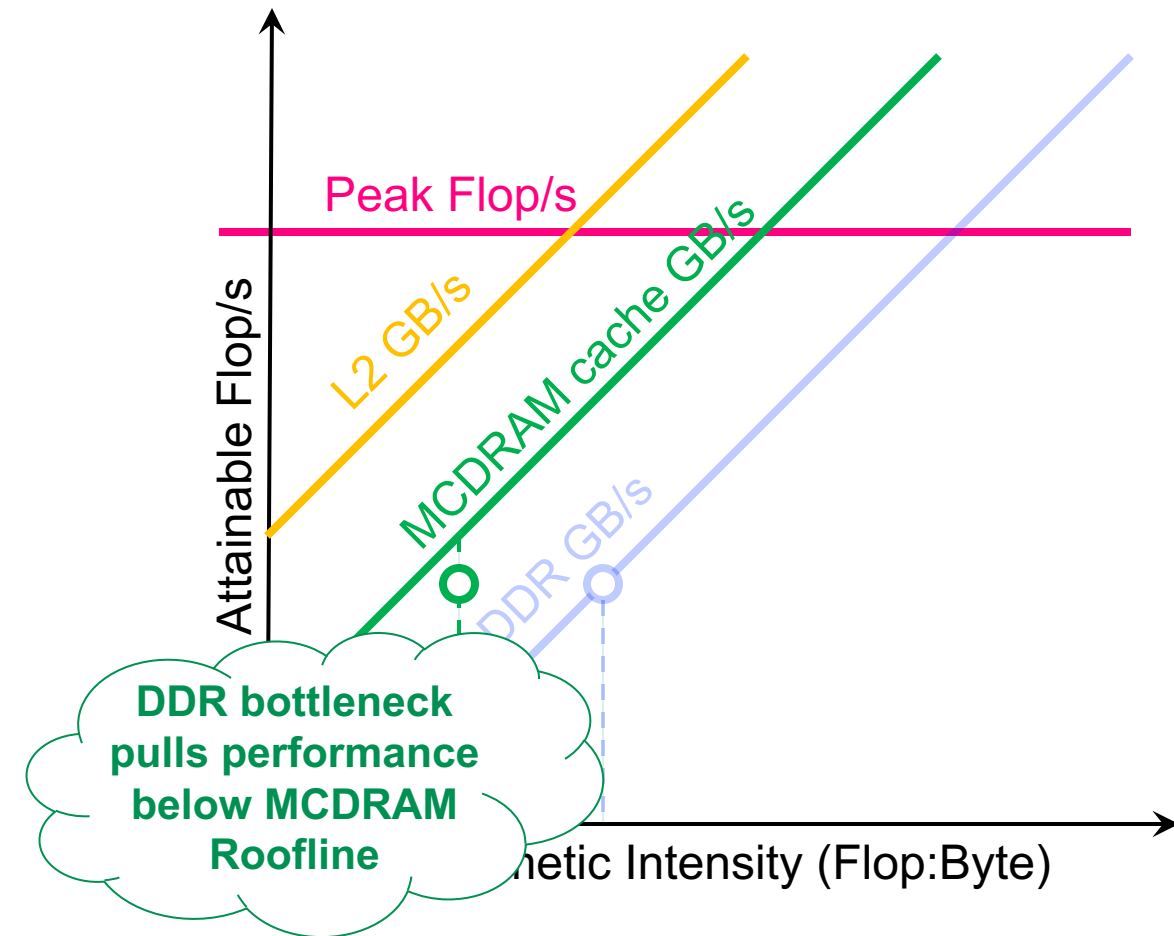
Hierarchical Roofline

- Construct superposition of Rooflines...
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, ... DRAM)...
 - ... **performance is bound by the minimum**



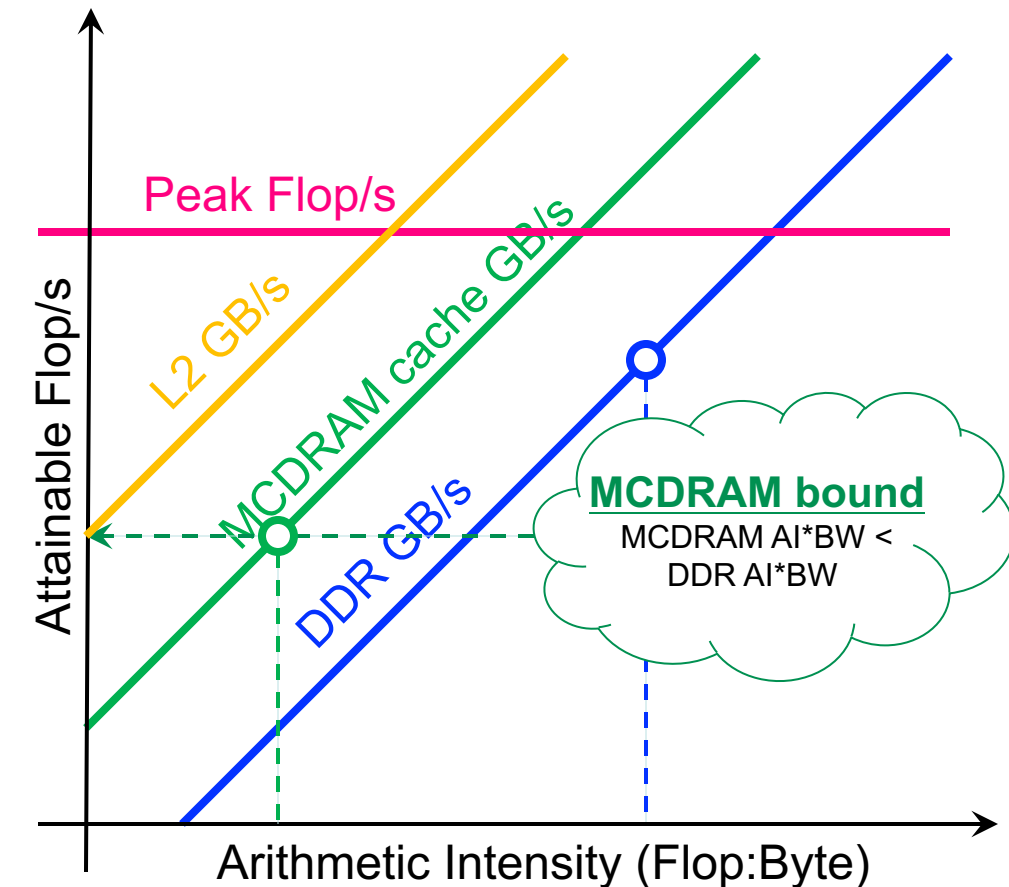
Hierarchical Roofline

- Construct superposition of Rooflines...
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, ... DRAM)...
 - ... **performance is bound by the minimum**



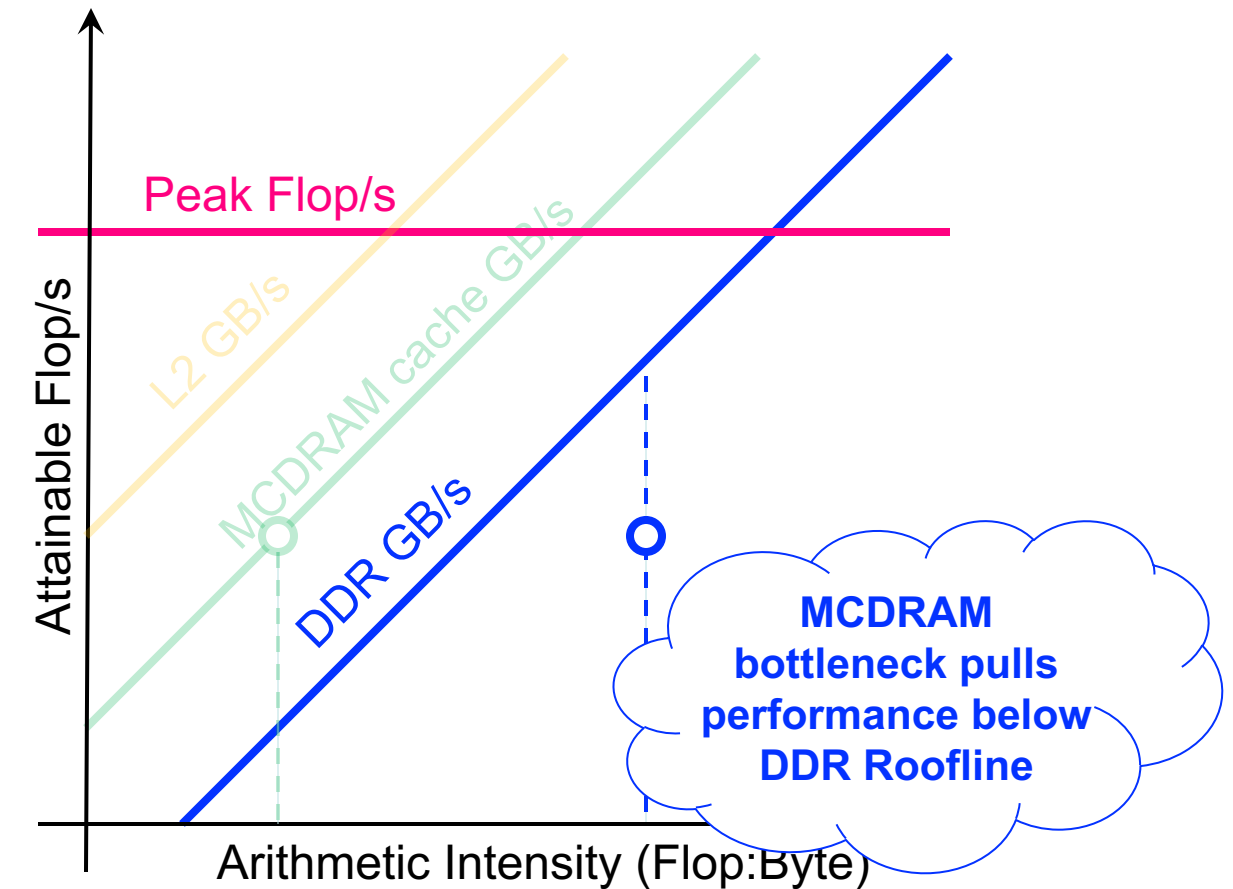
Hierarchical Roofline

- Construct superposition of Rooflines...
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, ... DRAM)...
 - ... **performance is bound by the minimum**



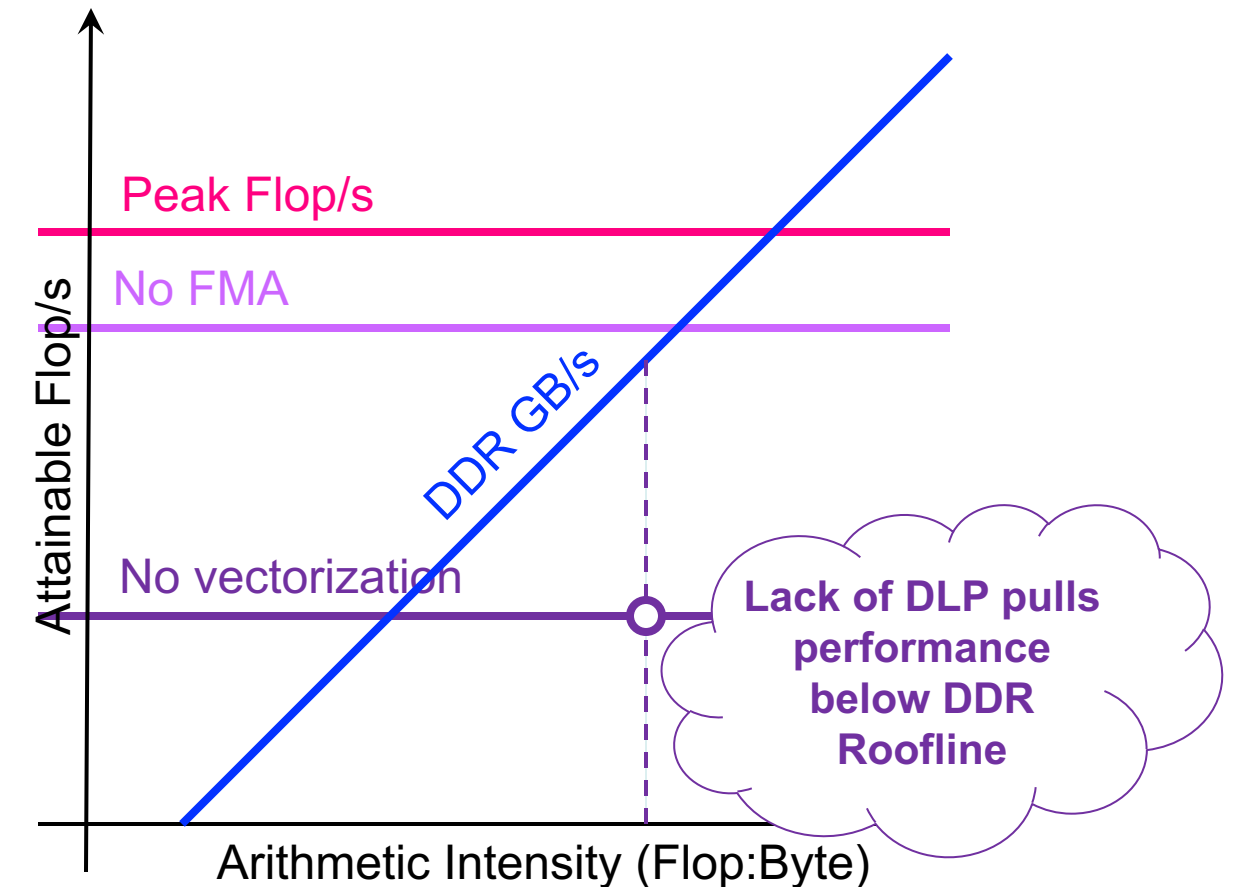
Hierarchical Roofline

- Construct superposition of Rooflines...
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, ... DRAM)...
 - ... **performance is bound by the minimum**



Data, Instruction, Thread-Level Parallelism...

- We have assumed one can attain peak flops with high locality.
- In reality, this is premised on sufficient...
 - Use special instructions (e.g. fused multiply-add)
 - Vectorization (16 flops per instruction)
 - unrolling, out-of-order execution (hide FPU latency)
 - OpenMP across multiple cores
- Without these, ...
 - Peak performance is not attainable
 - Some kernels can transition from memory-bound to compute-bound
 - n.b. in reality, DRAM bandwidth is often tied to DLP and TLP (single core can't saturate BW w/scalar code)



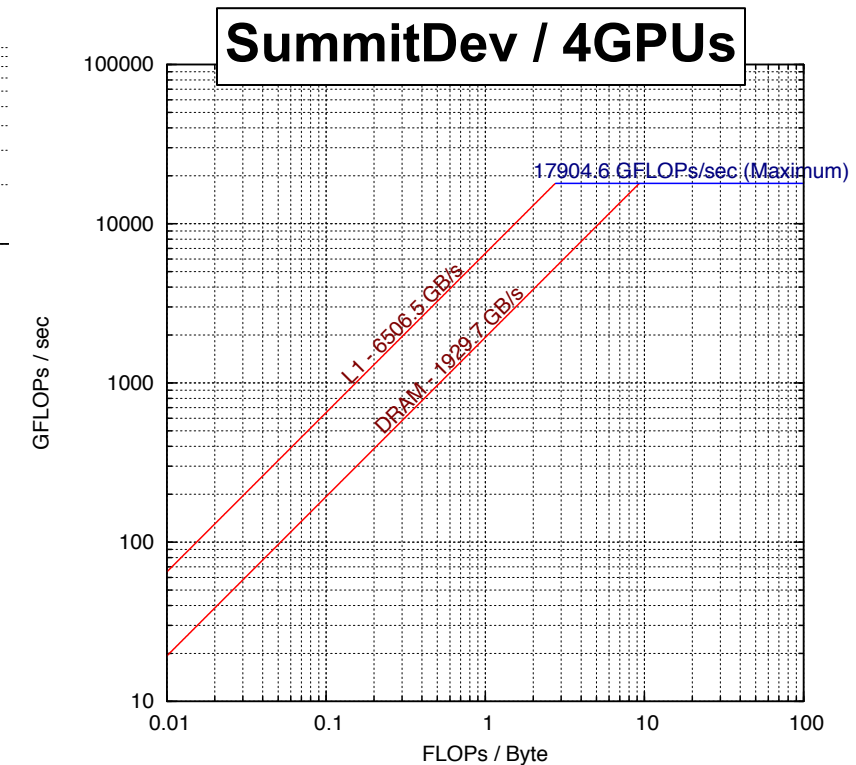
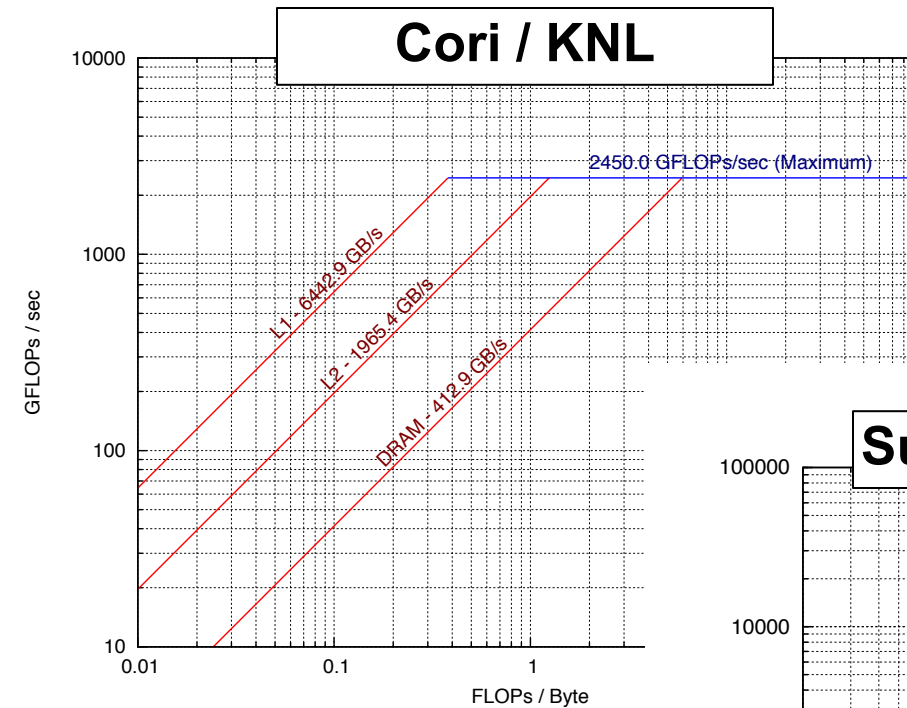
Initial LBL/NERSC Roofline Efforts

Initial LBL Roofline Efforts / Goals

1. Node Characterization
2. Application Instrumentation/Characterization
3. Using Roofline to drive application performance analysis and optimization for KNL.

Node Characterization?

- “Marketing Numbers” can be deceptive...
 - TurboMode / Underclock for AVX
 - Pin BW vs. real bandwidth
 - compiler failings on high-AI loops.
- LBL developed the Empirical Roofline Toolkit (ERT)...
 - Characterize CPU/GPU systems
 - Peak Flop rates
 - Bandwidths for each level of memory
 - **MPI+OpenMP/CUDA == multiple GPUs**



Instrumentation with Performance Counters?

- ***Characterizing applications with performance counters can be problematic...***
 - ✗ Flop Counters can be broken/missing in production processors
 - ✗ Vectorization/Masking can complicate counting Flop's
 - ✗ Counting Loads and Stores doesn't capture cache reuse while counting cache misses doesn't account for prefetchers.
 - ✗ DRAM counters (Uncore PMU) might be accurate, but...
 - are privileged and thus nominally inaccessible in user mode
 - may need vendor (e.g. Cray) and center (e.g. NERSC) approved OS/kernel changes

Forced to Cobble Together Tools...

- Use tools known/observed to work on NERSC's Cori (KNL, HSW)...
 - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
 - Used **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
- Accurate measurement of Flop's (HSW) and DRAM data movement (HSW and KNL)
- Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori...



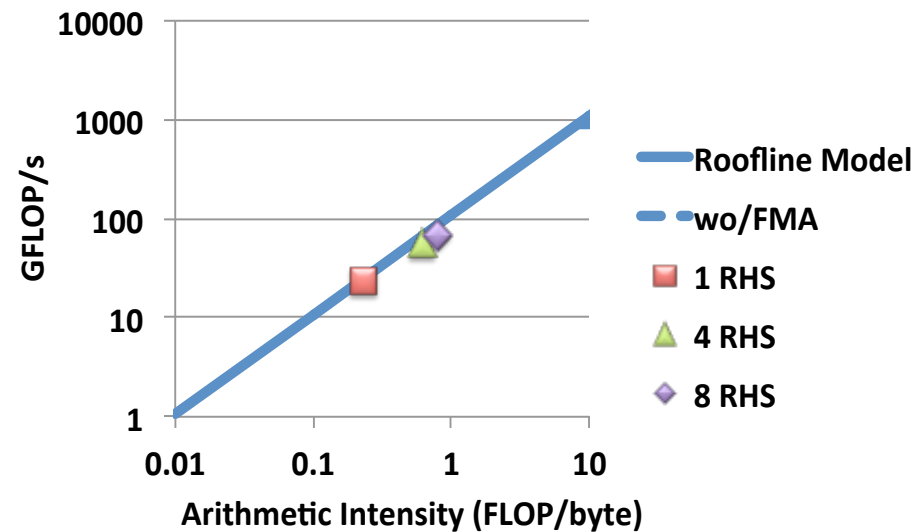
<http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/>

Initial Roofline Analysis of NESAP Codes

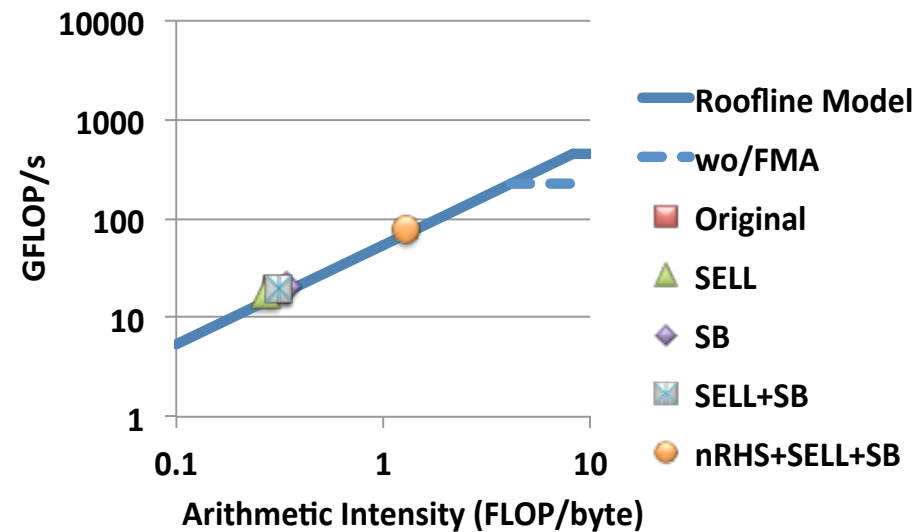
2P HSW

KNL

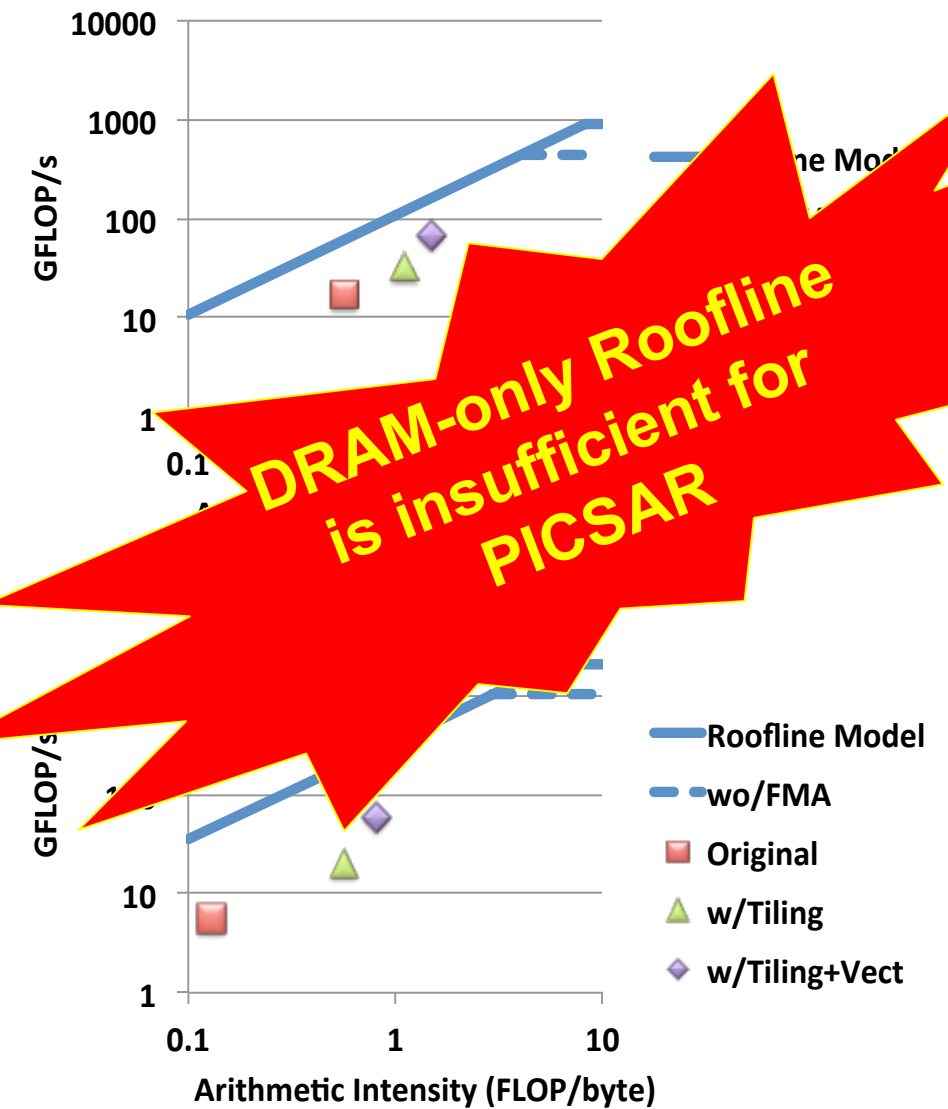
MFDn



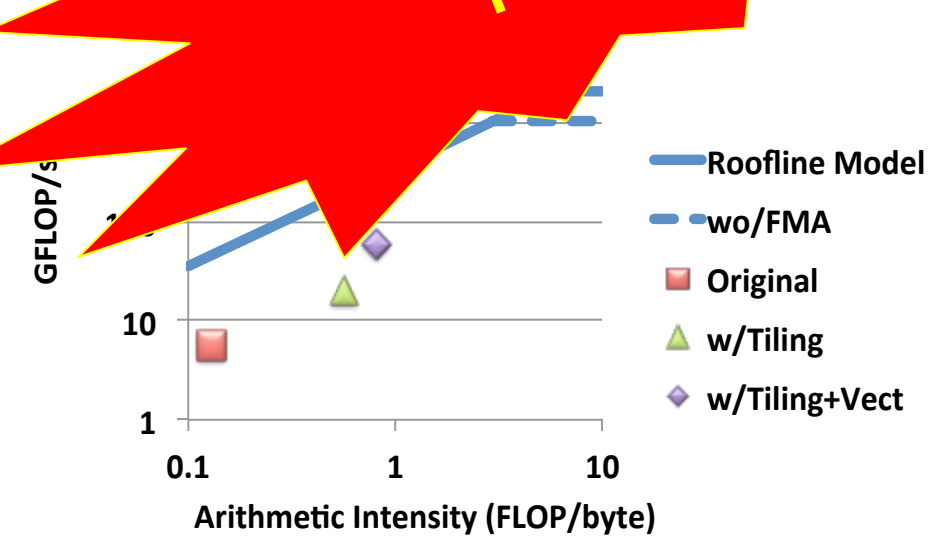
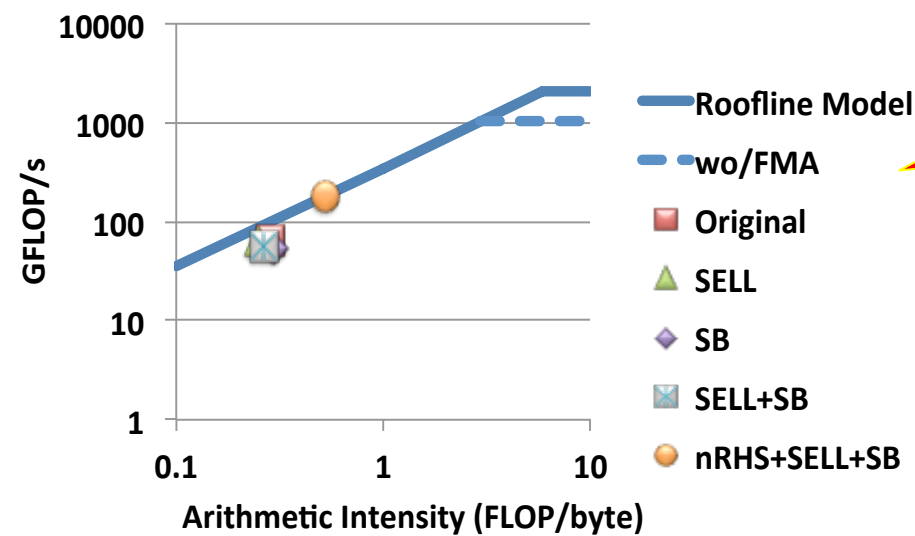
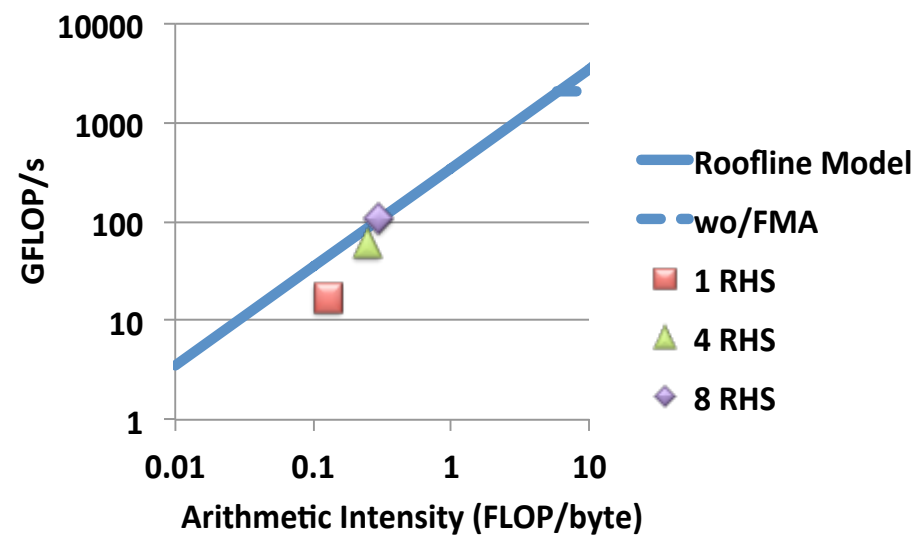
EMGeo



PICSAR

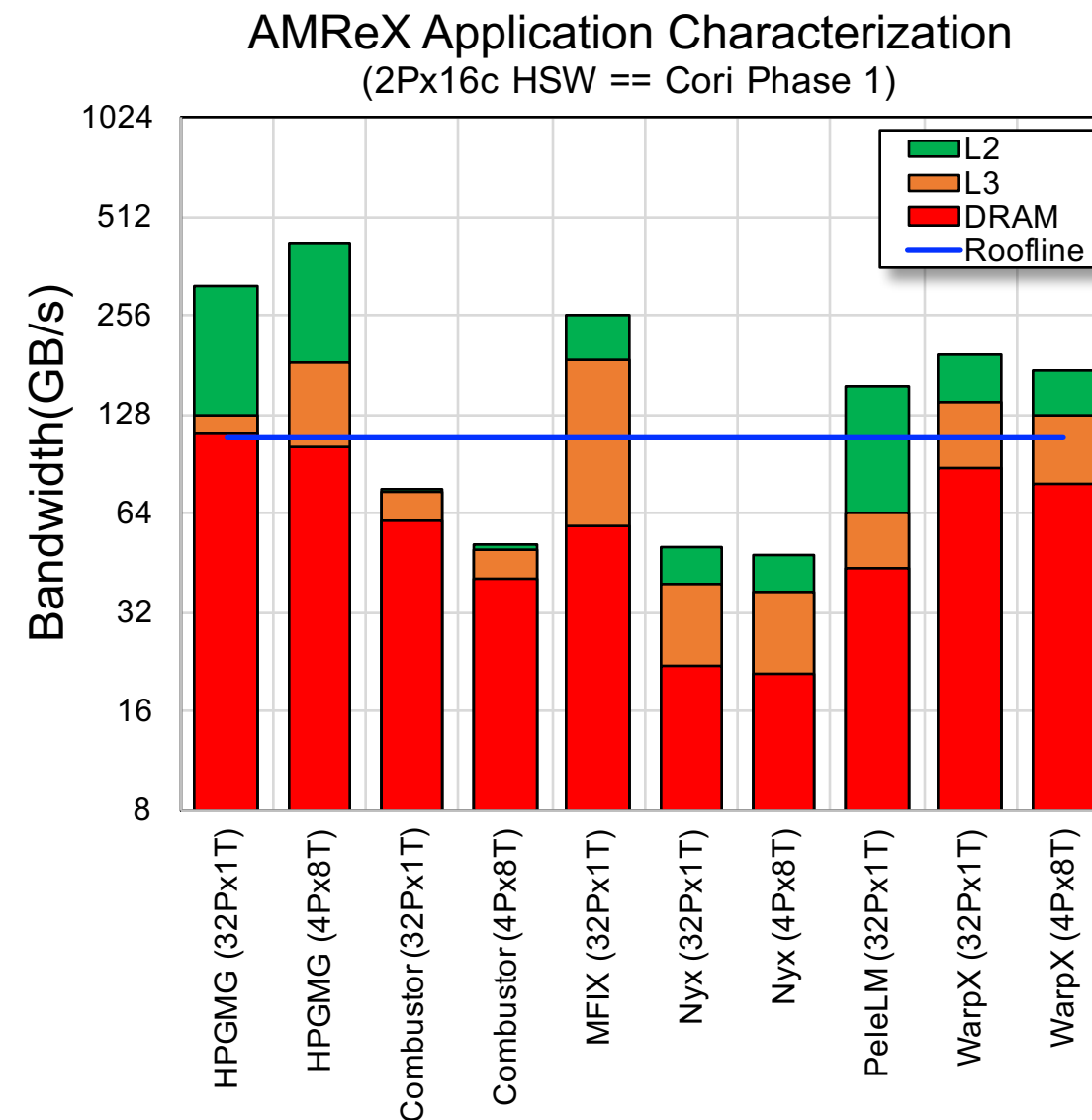


DRAM-only Roofline is insufficient for PICSAR



Evaluation of LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters...
 - ✓ Works on NERSC production systems
 - ✓ Minimal overhead (<1%)
 - ✓ Scalable in distributed memory (MPI-friendly)
 - ✓ Fast, high-level characterization
 - ✗ No detailed timing breakdown or optimization advice
 - ✗ Limited by quality of hardware performance counter implementation (garbage in/garbage out)
- Useful tool that complements other tools



Need an integrated solution...

- Having to compose VTune, SDE, and plotting tools...
 - ✓ worked correctly and benefited NESAP's application readiness
 - ✗ forced users to learn/run multiple tools and manually parse/graph the output
 - ✗ forced users to instrument routines of interest in their application
 - ✗ lacked integration with compiler/debugger/disassembly
- LIKWID was...
 - ✓ fast and easy to use
 - ✗ Suffered from the same limitations as VTune/SDE
- ERT...
 - ✓ Characterized flops, and bandwidths (cache, DRAM)
 - ✓ Interoperable with MPI, OpenMP, and CUDA
 - ✗ Required users to manually parse/incorporate the output

Intel Advisor

- Includes Roofline Automation...
 - ✓ Automatically instruments applications (one dot per loop nest/function)
 - ✓ Computes FLOPS and AI for each function (**CARM**)
 - ✓ Full AVX-512 integration that incorporates mask values
 - ✓ **Integrated Cache Simulator¹** (hierarchical roofline / multiple AI's)
 - ✓ Automatically benchmarks target system (calculates ceilings)
 - ✓ Full integration with existing Advisor capabilities



¹Technology Preview, not in official product roadmap so far.
This version will be made available during the hands-on component of this tutorial.

Hierarchical Roofline vs. Cache-Aware Roofline

*...understanding different Roofline
formulations in Advisor*

There are two Major Roofline Formulations:

- Hierarchical Roofline (original Roofline w/ DRAM, L3, L2, ...)...
 - Williams, et al, “Roofline: An Insightful Visual Performance Model for Multicore Architectures”, CACM, 2009
 - Chapter 4 of “Auto-tuning Performance on Multicore Computers”, 2008
 - Defines multiple bandwidth ceilings and multiple AI’s per kernel
 - Performance bound is the minimum of flops and the memory intercepts (superposition of original, single-metric Rooflines)
- Cache-Aware Roofline
 - Ilic et al, "Cache-aware Roofline model: Upgrading the loft", IEEE Computer Architecture Letters, 2014
 - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
 - As one loses cache locality (capacity, conflict, ...) performance falls from one BW ceiling to a lower one at constant AI
- Why Does this matter?
 - Some tools use the Hierarchical Roofline, some use cache-aware == **Users need to understand the differences**
 - Cache-Aware Roofline model was integrated into production Intel Advisor
 - Evaluation version of Hierarchical Roofline¹ (cache simulator) has also been integrated into Intel Advisor
 - **You will be allowed to explore both in the hand—on component of this tutorial**

¹Technology Preview, not in official product roadmap so far.
This version will be made available during the hands-on component of this tutorial.

Hierarchical Roofline

- Captures cache effects
- AI is Flop:Bytes after being *filtered by lower cache levels*
- Multiple Arithmetic Intensities (one per level of memory)
- AI *dependent* on problem size (capacity misses reduce AI)
- Memory/Cache/Locality effects are *observed as decreased AI*
- Requires *performance counters or cache simulator* to correctly measure AI

Cache-Aware Roofline

- Captures cache effects
- AI is Flop:Bytes *as presented to the L1 cache (plus non-temporal stores)*
- Single Arithmetic Intensity
- AI *independent* of problem size
- Memory/Cache/Locality effects are *observed as decreased performance*
- Requires static analysis or *binary instrumentation* to measure AI

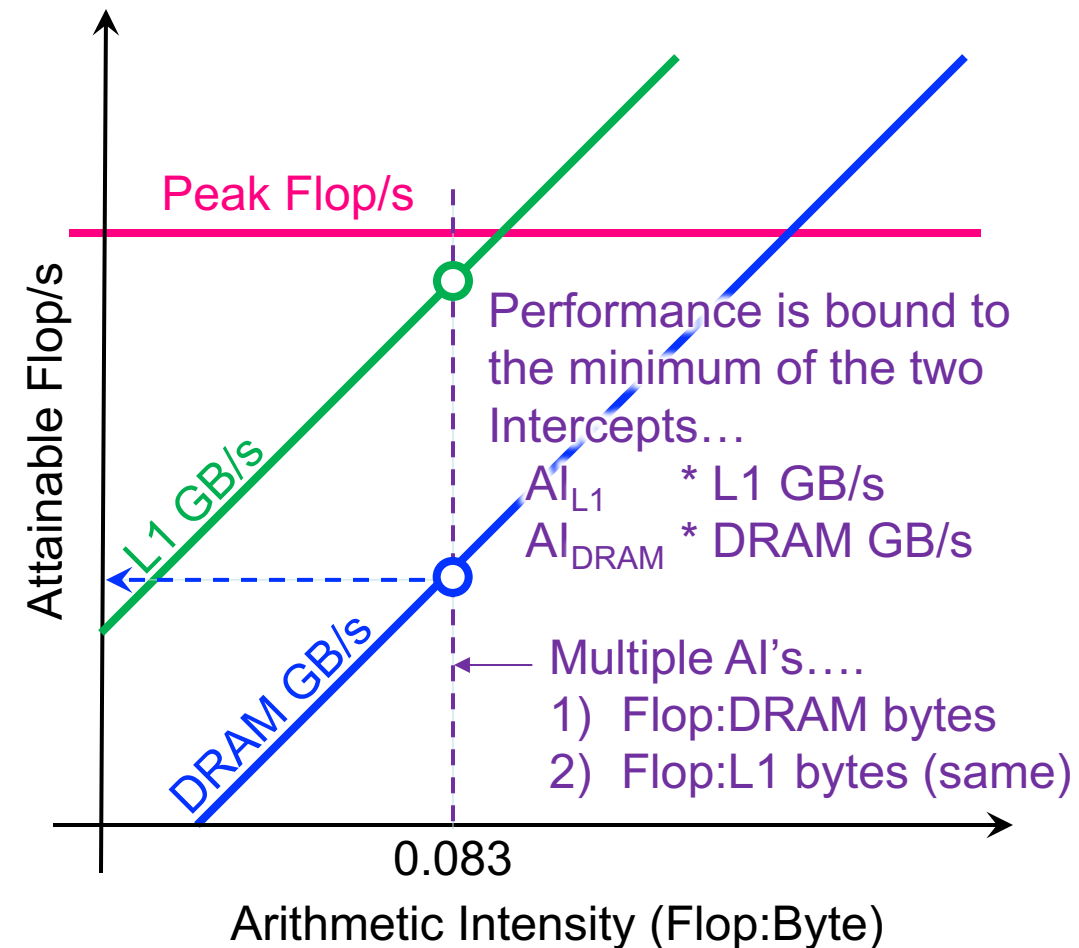
Example: STREAM

- L1 AI...
 - 2 flops
 - 2 x 8B load (old)
 - 1 x 8B store (new)
 - = 0.08 flops per byte
- No cache reuse...
 - Iteration i doesn't touch any data associated with iteration $i + \text{delta}$ for any delta .
- ... leads to a DRAM AI equal to the L1 AI

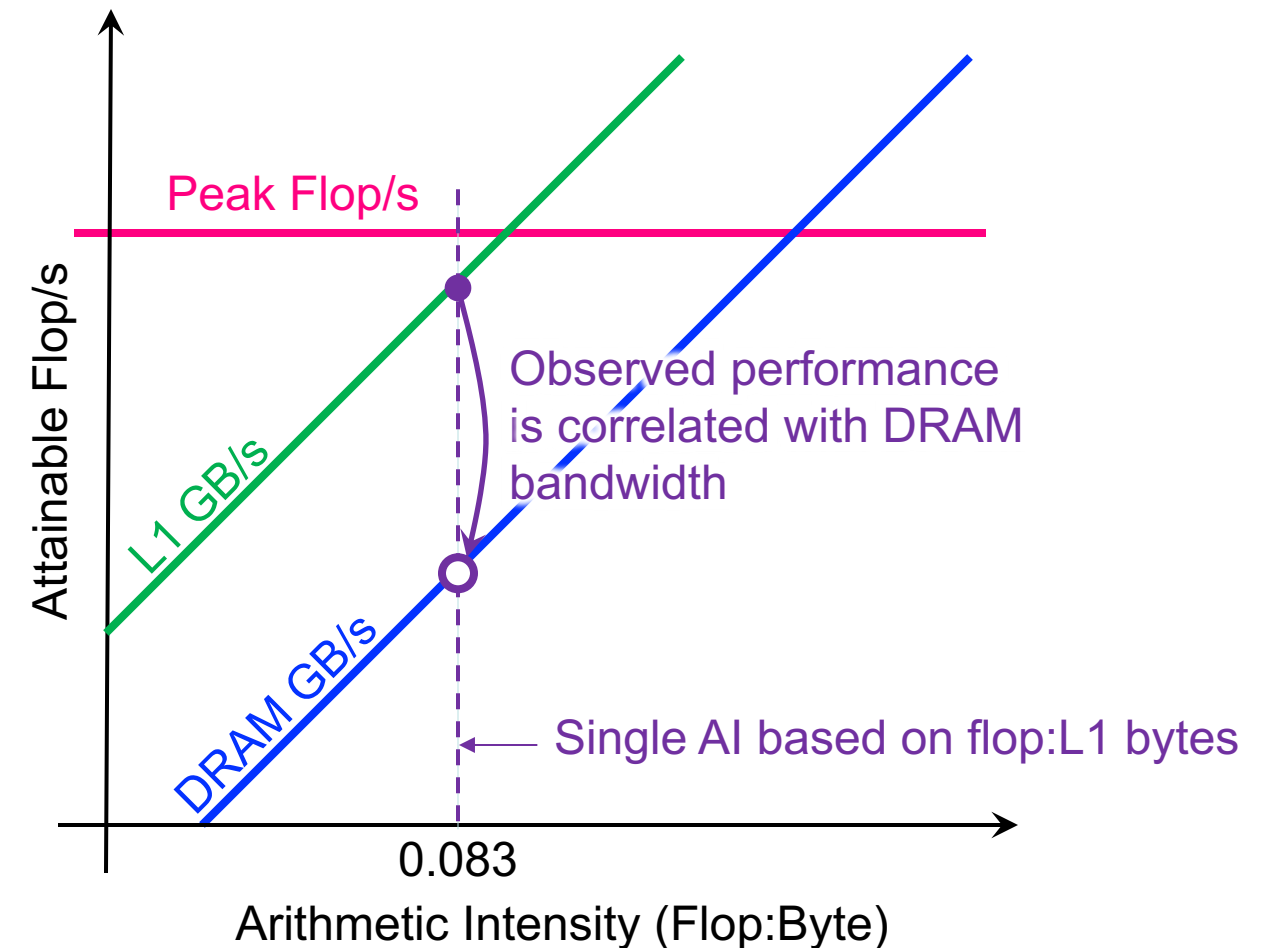
```
#pragma omp parallel for  
for(i=0;i<N;i++){  
    z[i] = x[i] + alpha*y[i];  
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



Example: 7-point Stencil (Small Problem)

■ L1 AI...

- 7 flops
- 7 x 8B load (old)
- 1 x 8B store (new)
- = 0.11 flops per byte
- some compilers may do register shuffles to reduce the number of loads.

■ Moderate cache reuse...

- `old[ijk]` is reused on subsequent iterations of `i,j,k`
- `old[ijk-1]` is reused on subsequent iterations of `i`.
- `old[ijk-jStride]` is reused on subsequent iterations of `j`.
- `old[ijk-kStride]` is reused on subsequent iterations of `k`.

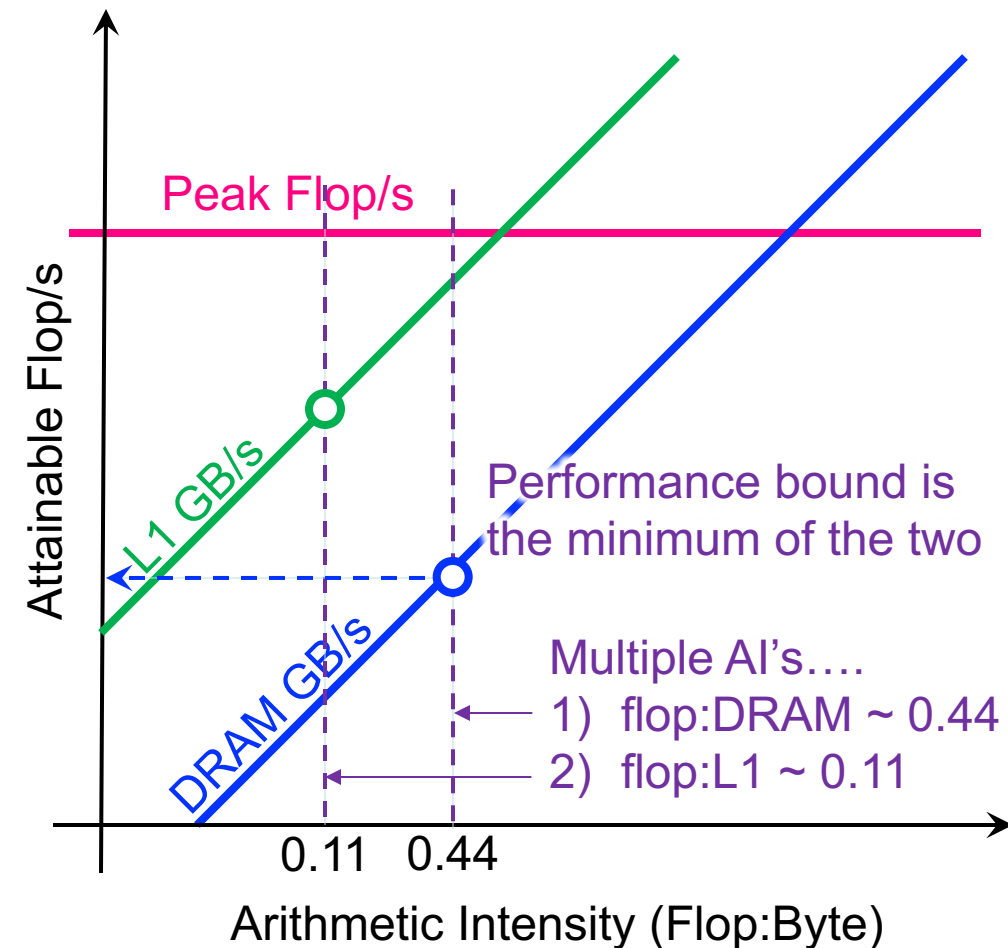
■ ... leads to DRAM AI larger than the L1 AI

```
#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++){
    int ijk = i + j*jStride + k*kStride;
    new[ijk] = -6.0*old[ijk
                        + old[ijk-1
                        + old[ijk+1
                        + old[ijk-jStride]
                        + old[ijk+jStride]
                        + old[ijk-kStride]
                        + old[ijk+kStride];
}}}

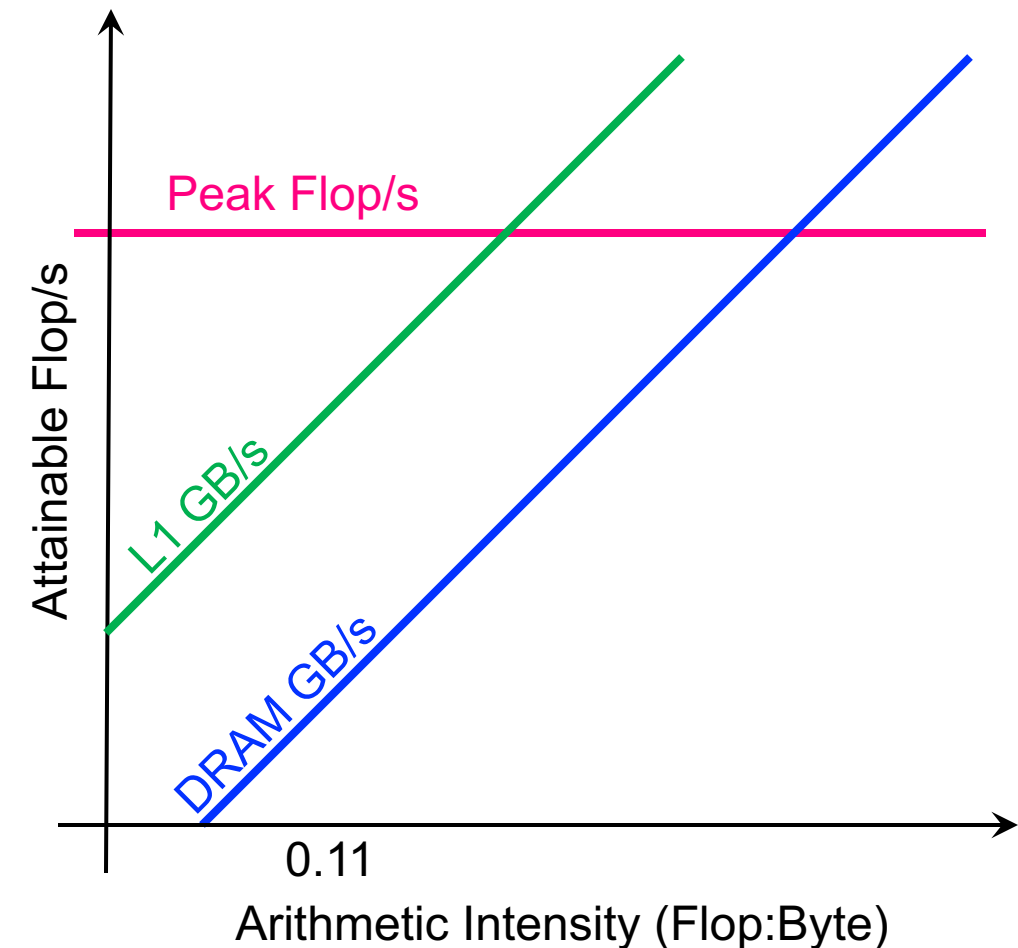
```

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

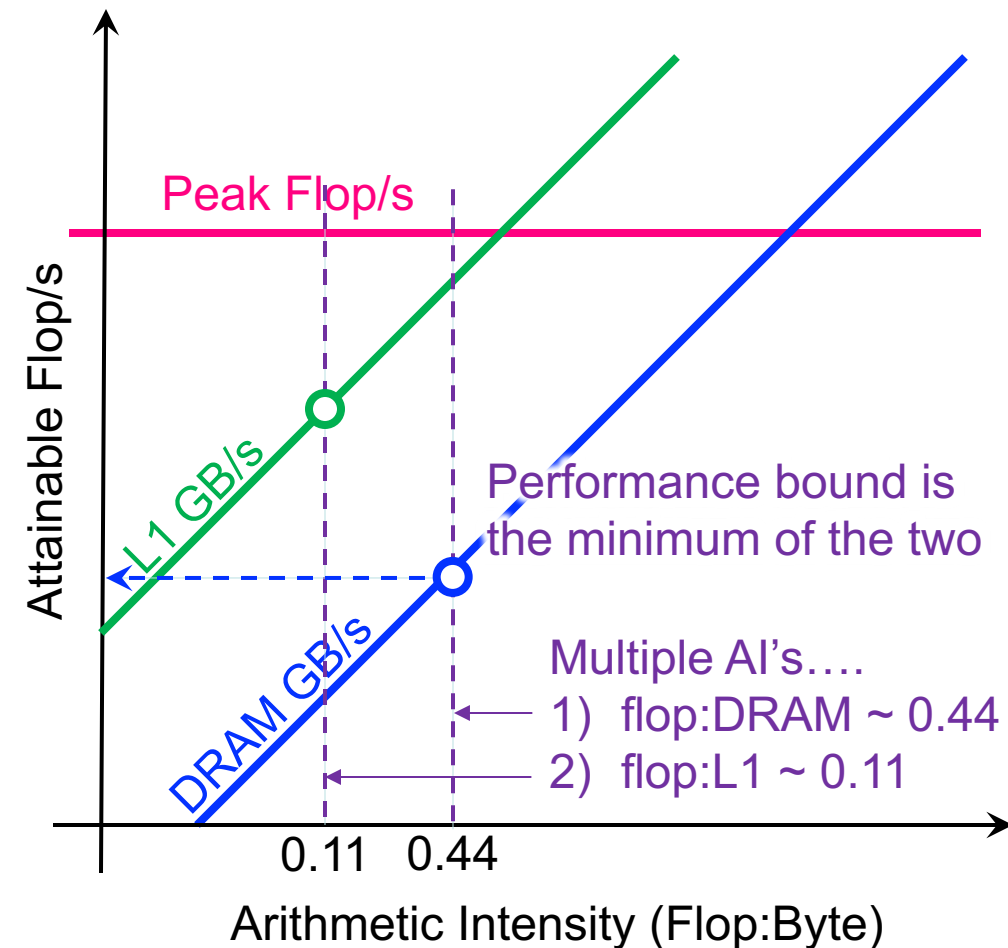


Cache-Aware Roofline

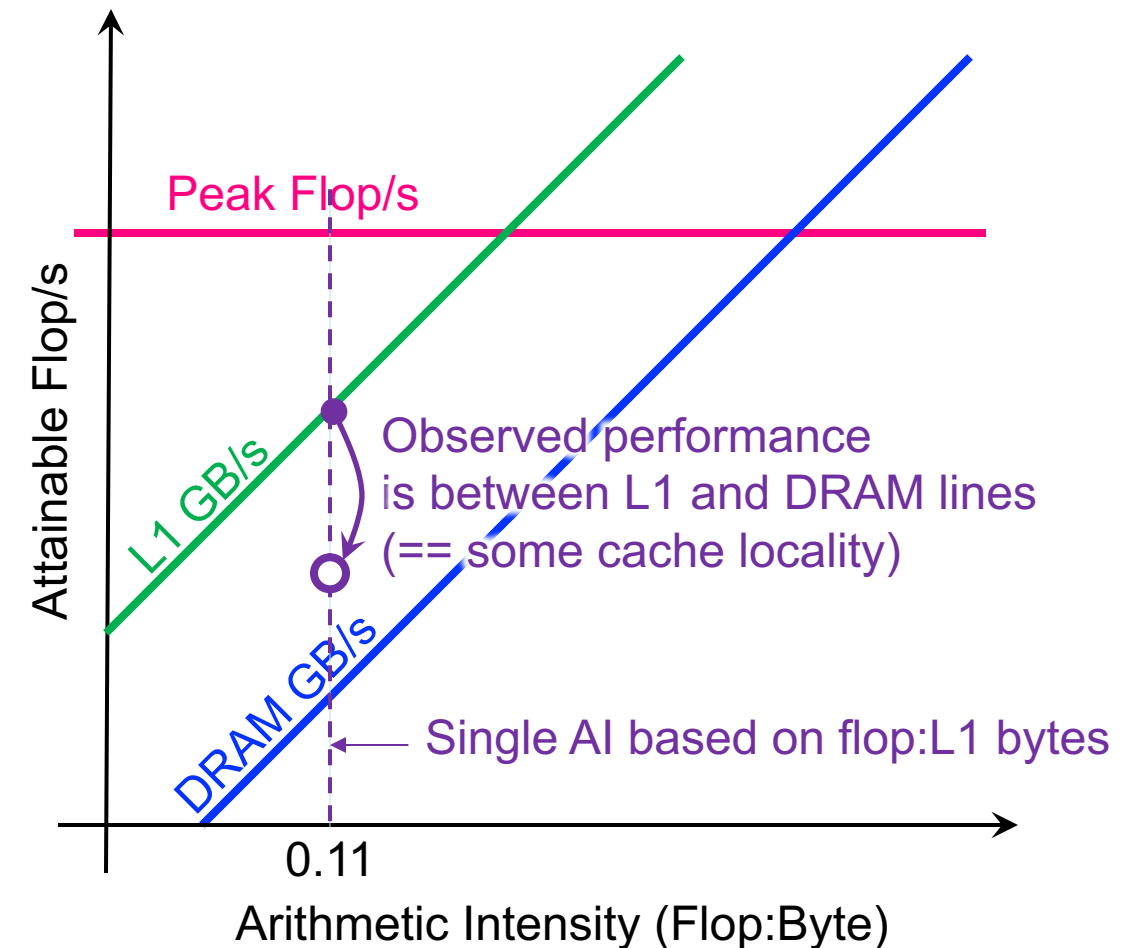


Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

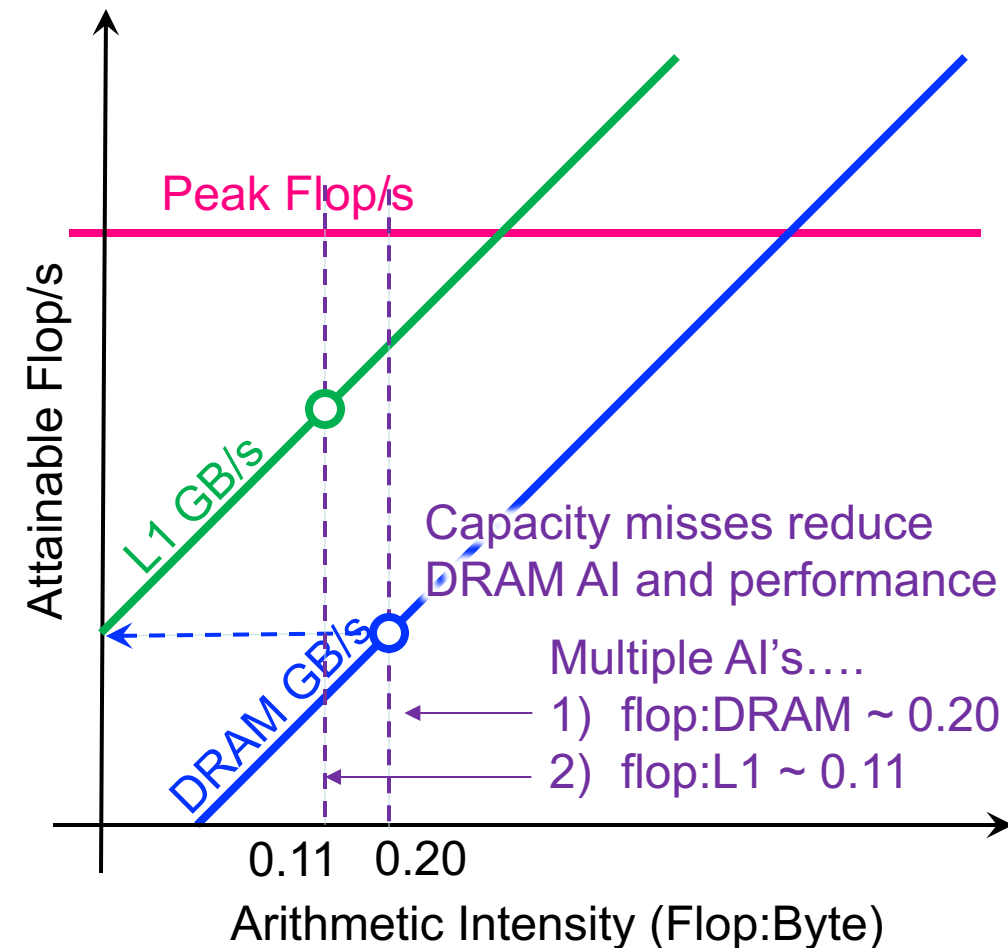


Cache-Aware Roofline

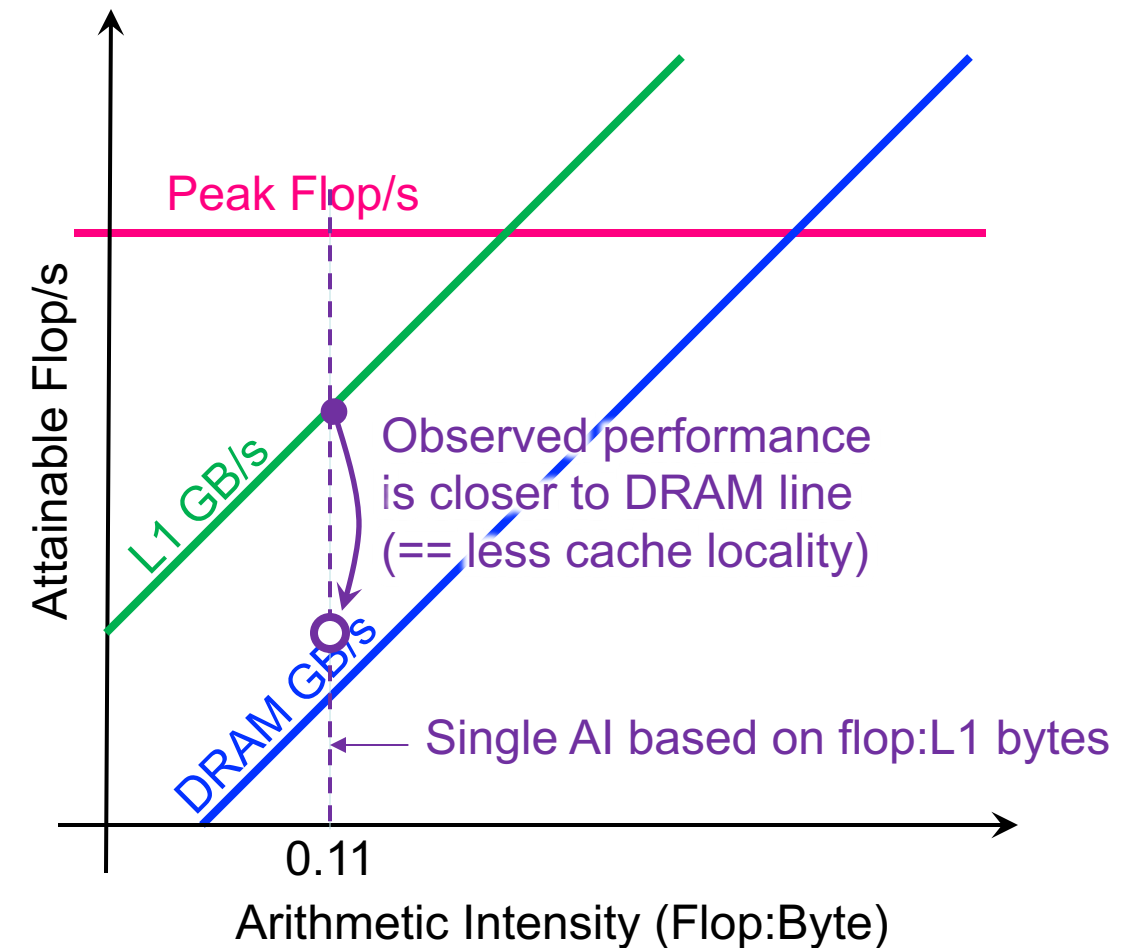


Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

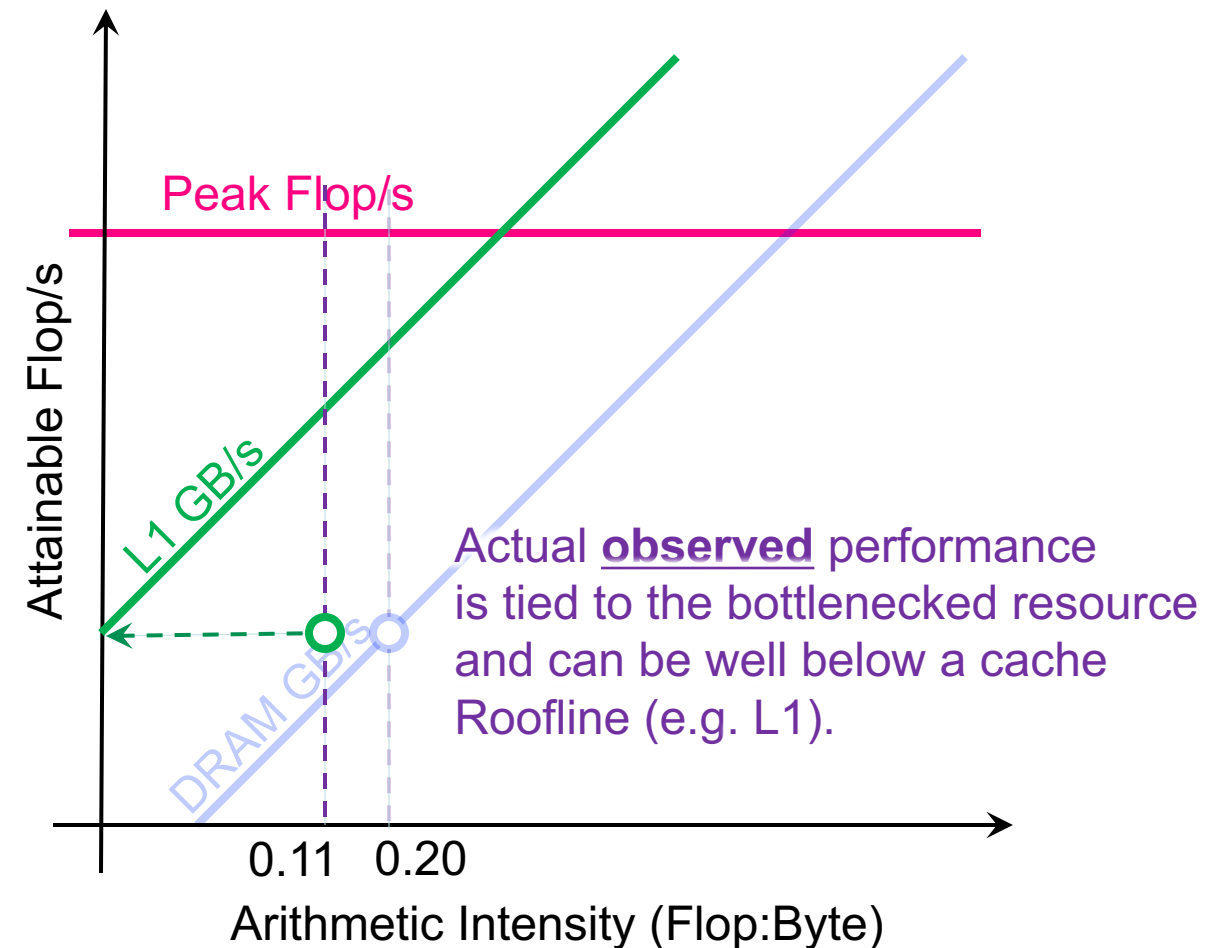


Cache-Aware Roofline

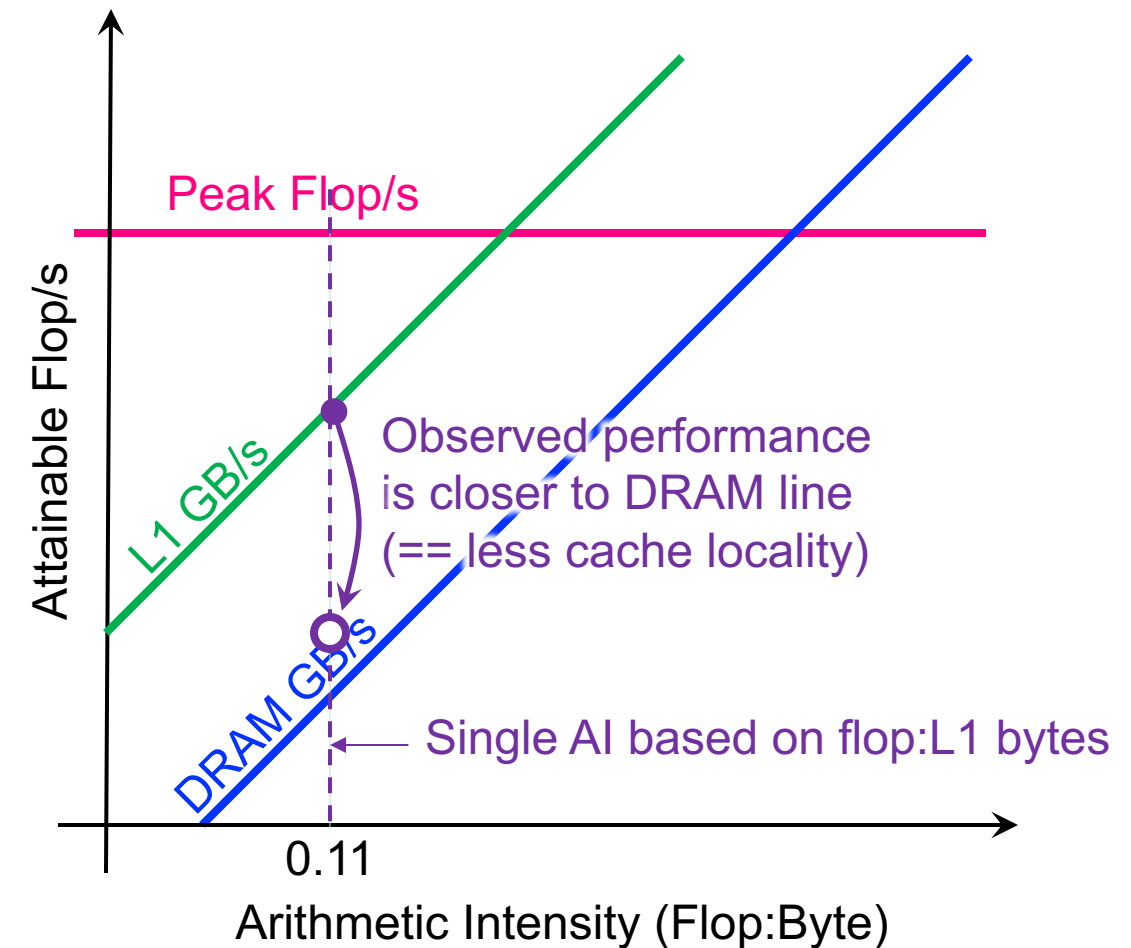


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

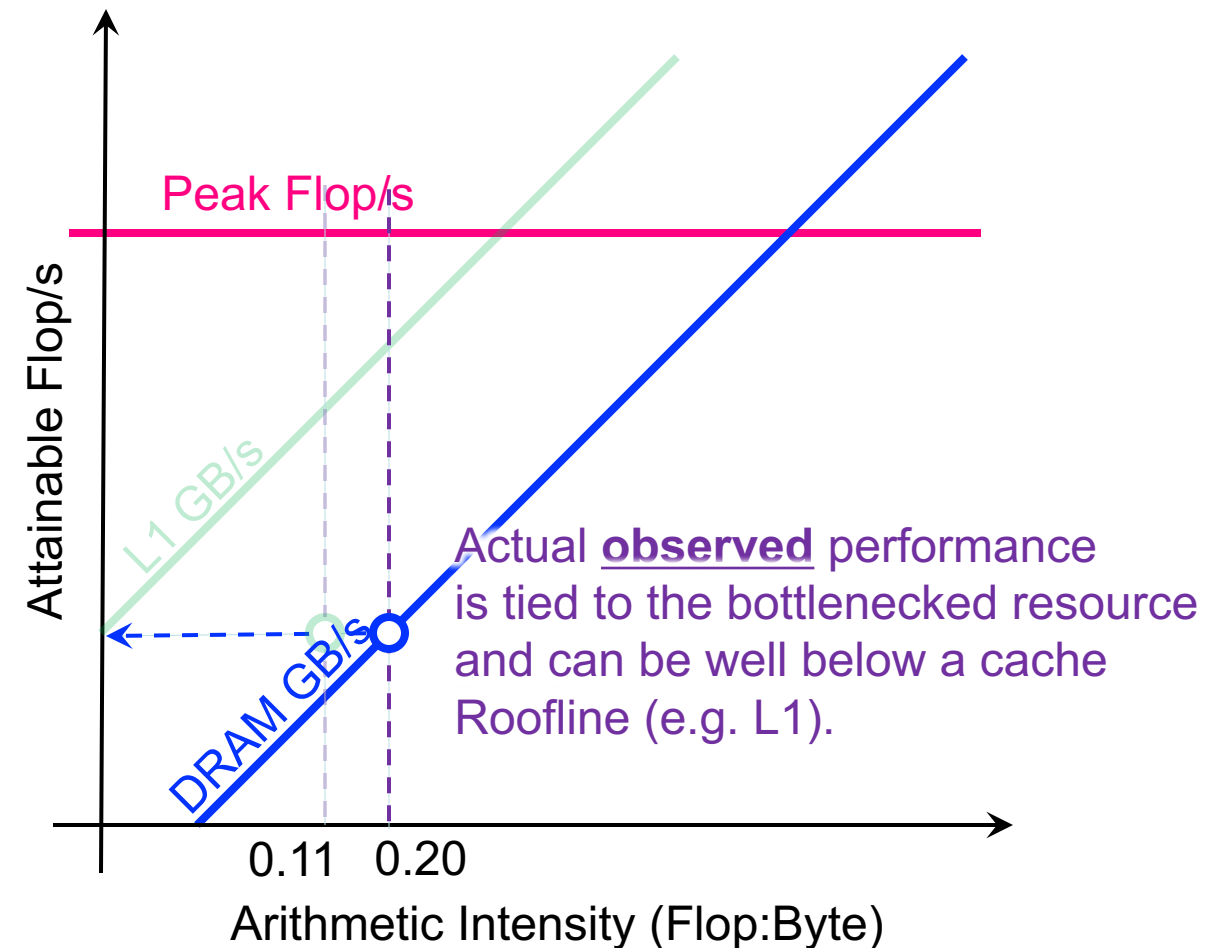


Cache-Aware Roofline

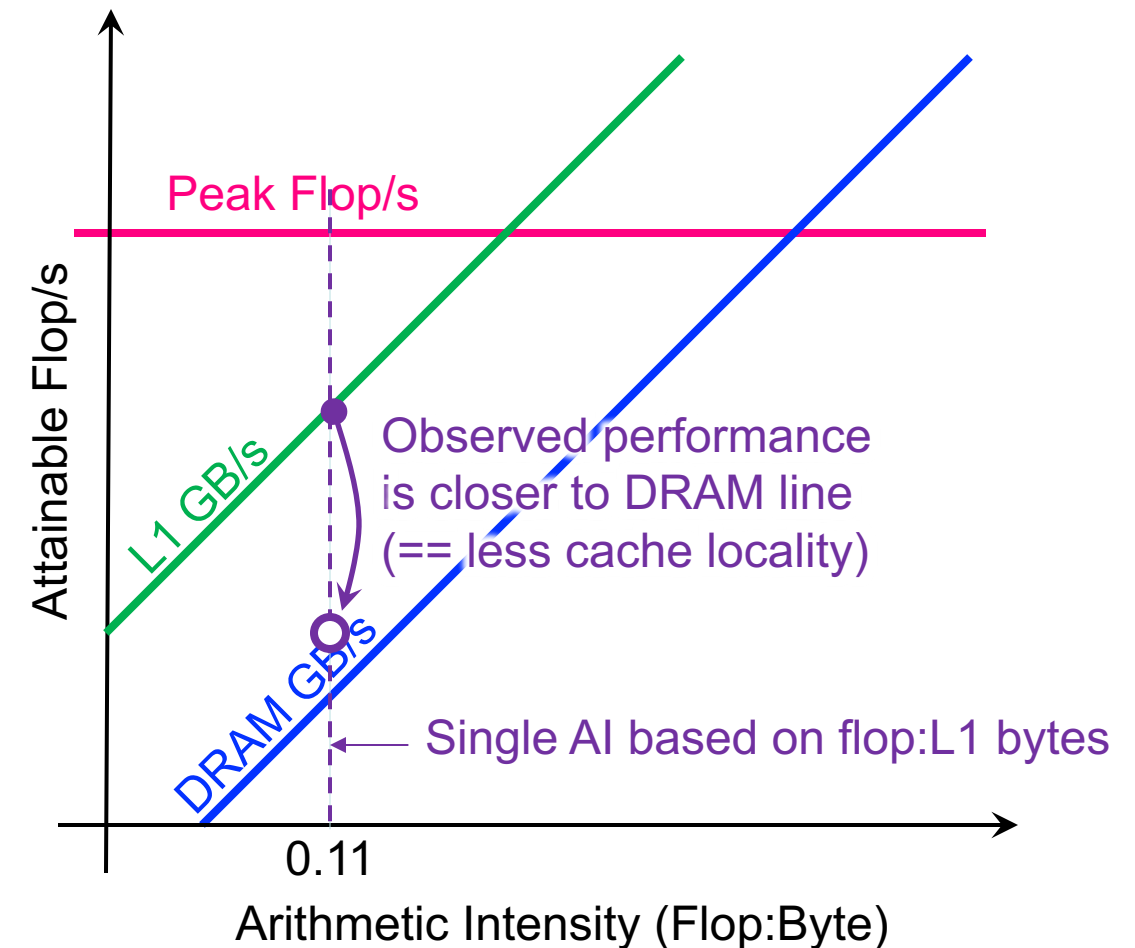


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Cache-Aware Roofline



Questions?