Performance Tuning of Scientific Codes with the Roofline Model

1:30pm	Introductions / Administration	all
1:35pm	Roofline Introduction	Samuel Williams
2:10pm	CARM / Energy / GPUs	Aleksandar Ilic
2:40pm	Intel Advisor Installation	Zakhar Matveev
3:00pm	coffee break	
3:30pm	Introduction to Intel Advisor	Zakhar Matveev
3:45pm	Hands-on with Intel Advisor	all
4:30pm	HPC Application Studies	Charlene Yang
4:55pm	closing remarks / Q&A	all

Introductions

Samuel Williams
Computational Research Division
Lawrence Berkeley National Lab
SWWilliams@lbl.gov

Aleksandar Ilic
Institute of Systems and Computer Engineering, Portugal
aleksandar.ilic@inesc-id.pt

Charlene Yang
NERSC
Lawrence Berkeley National Lab
CJYang@lbl.gov

Zakhar Matveev
Intel Corporation
zakhar.a.matveev@intel.com

Kiril Rogozhin
Intel Corporation
kirill.rogozhin@intel.com

Materials:

USB / Downloads

more Roofline at SC'18...

P3HPC Workshop	**Friday 8:30am D174**	**"An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability",** Yang, Gayatri, Kurth, Basu, Ronaghi, Adetokunbo, Friesen, Cook, Doerfler, Oliker, Deslippe, Williams

Don’t forget to take the Survey…

http://bit.ly/sc18-eval

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.

Background

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

- Many different components can contribute to kernel run time.
- Some are application-specific, and some architecture-specific.

#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

Performance Models

- Can’t think about all these terms all the time for every application…

Computational Complexity

#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait’s	Network Latency

Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

LogP

Culler, et al, "LogP: a practical model of parallel computation", CACM, 1996.

Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

LogGP

Alexandrov, et al, "LogGP: incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation", SPAA, 1995.

Performance Models

- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.

#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

Roofline Model

Williams et al, "Roofline: An Insightful Visual Performance Model For Multicore Architectures", CACM, 2009.

Roofline Model:

Arithmetic Intensity and Bandwidth

Performance Models / Simulators

- Historically, many performance models and simulators tracked time to predict performance (i.e. counting cycles)

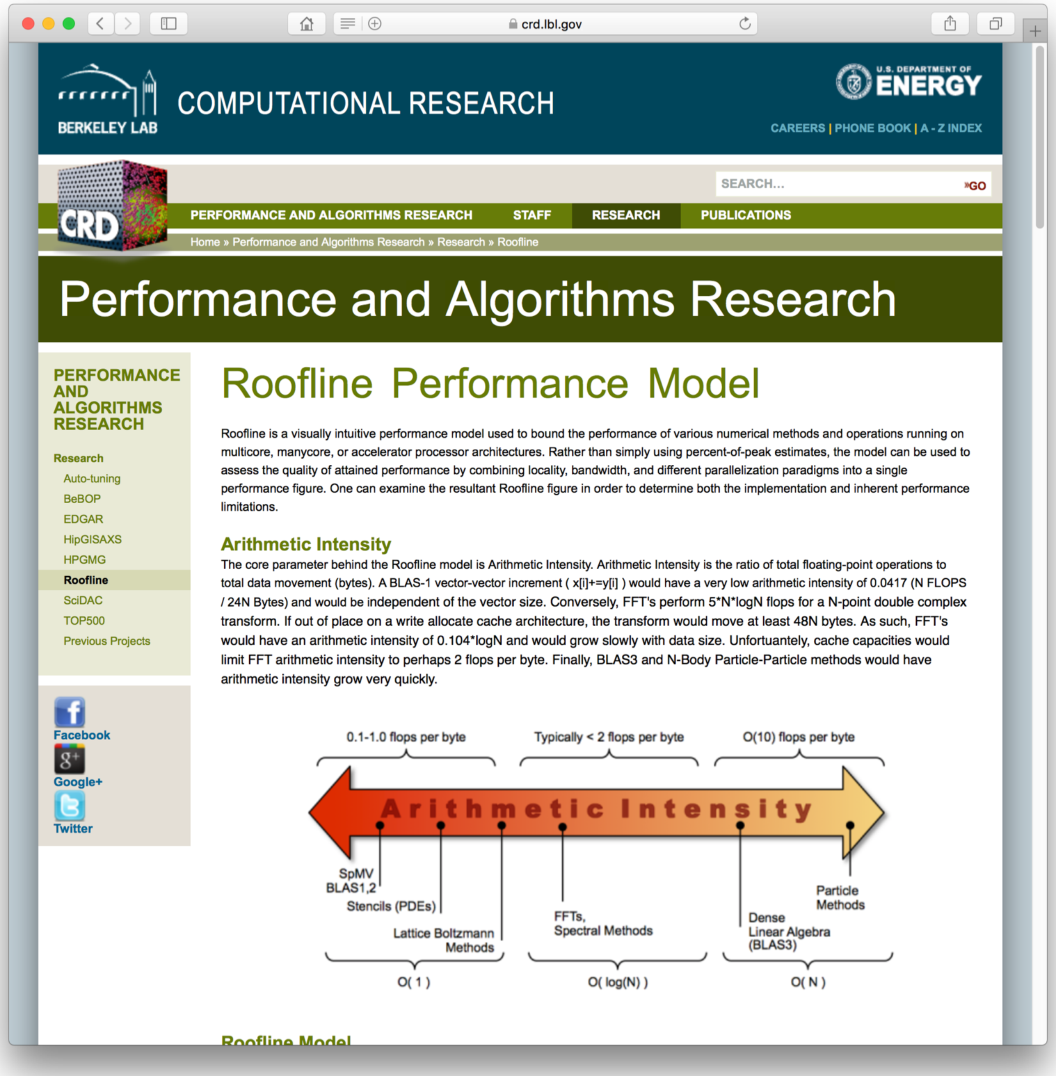
- The last two decades saw a number of latency-hiding techniques…
  - Out-of-order execution (hardware discovers parallelism to hide latency)
  - HW stream prefetching (hardware speculatively loads data)
  - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)

- … resulted in a shift from a latency-limited computing regime to a **throughput-limited computing regime**

Roofline Model

- **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 TPUs[1], etc…)



Performance and Algorithms Research

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, FFT's perform 5*N*logN 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*logN and would grow slowly with data size. Unfortuantely, 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.

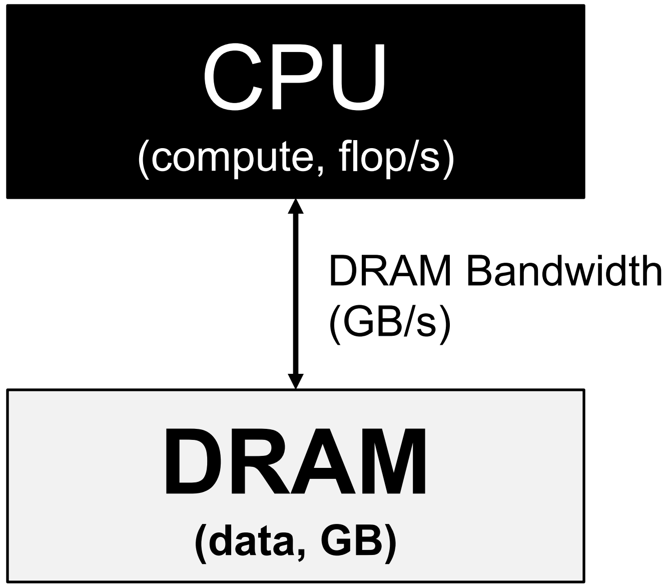
Roofline Model

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

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

(DRAM) Roofline

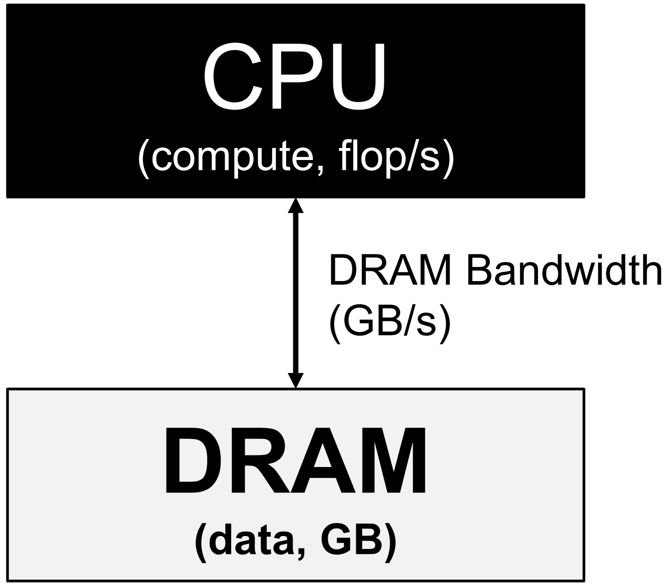
- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…



$$\text{Time} = \max\begin{cases}\text{\#FP ops / Peak GFlop/s}\\ \text{\#Bytes / Peak GB/s}\end{cases}$$

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…



CPU

(compute, flop/s)

DRAM Bandwidth (GB/s)

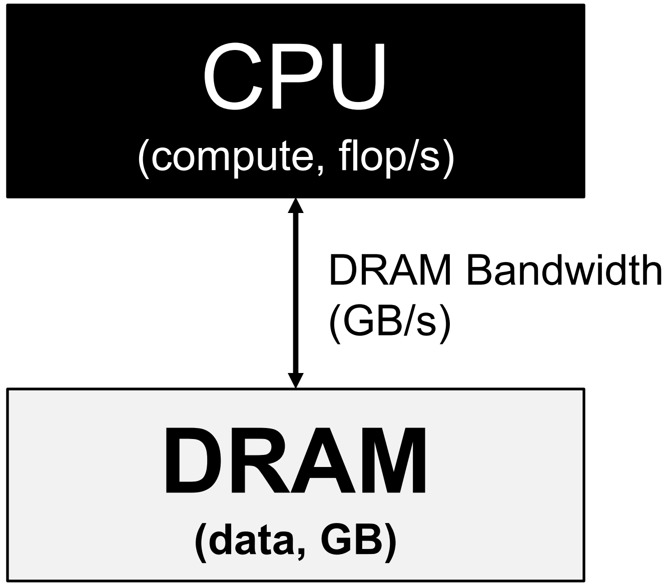
DRAM

(data, GB)

$$\frac{\text{Time}}{\text{\#FP ops}} = \max \begin{cases} \text{1 / Peak GFlop/s} \\ \text{\#Bytes / \#FP ops / Peak GB/s} \end{cases}$$

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…



CPU

(compute, flop/s)

DRAM Bandwidth
(GB/s)

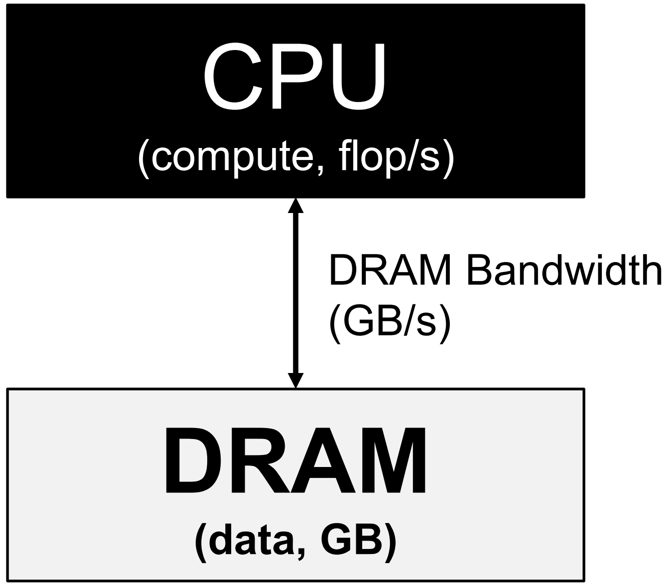
DRAM

(data, GB)

$$\frac{\#\text{FP ops}}{\text{Time}} = \min\begin{cases}\text{Peak GFlop/s} \\ (\#\text{FP ops} / \#\text{Bytes}) * \text{Peak GB/s}\end{cases}$$

(DRAM) Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite reuse and bandwidth limit performance.
- Assuming perfect overlap of communication and computation…



CPU

(compute, flop/s)

DRAM Bandwidth (GB/s)

DRAM

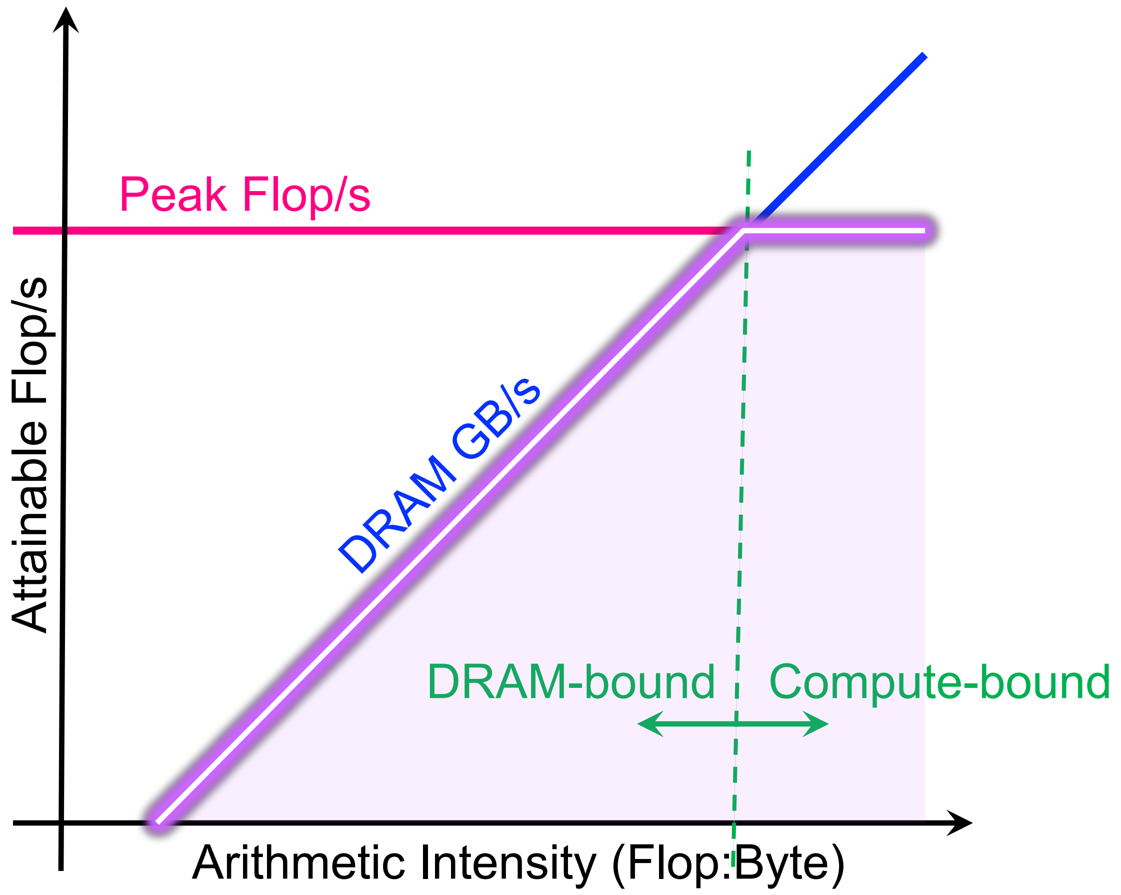
(data, GB)

$$\textbf{GFlop/s} = \textbf{min} \begin{cases} \textbf{Peak GFlop/s} \\ \textbf{AI * Peak GB/s} \end{cases}$$

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

(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…)



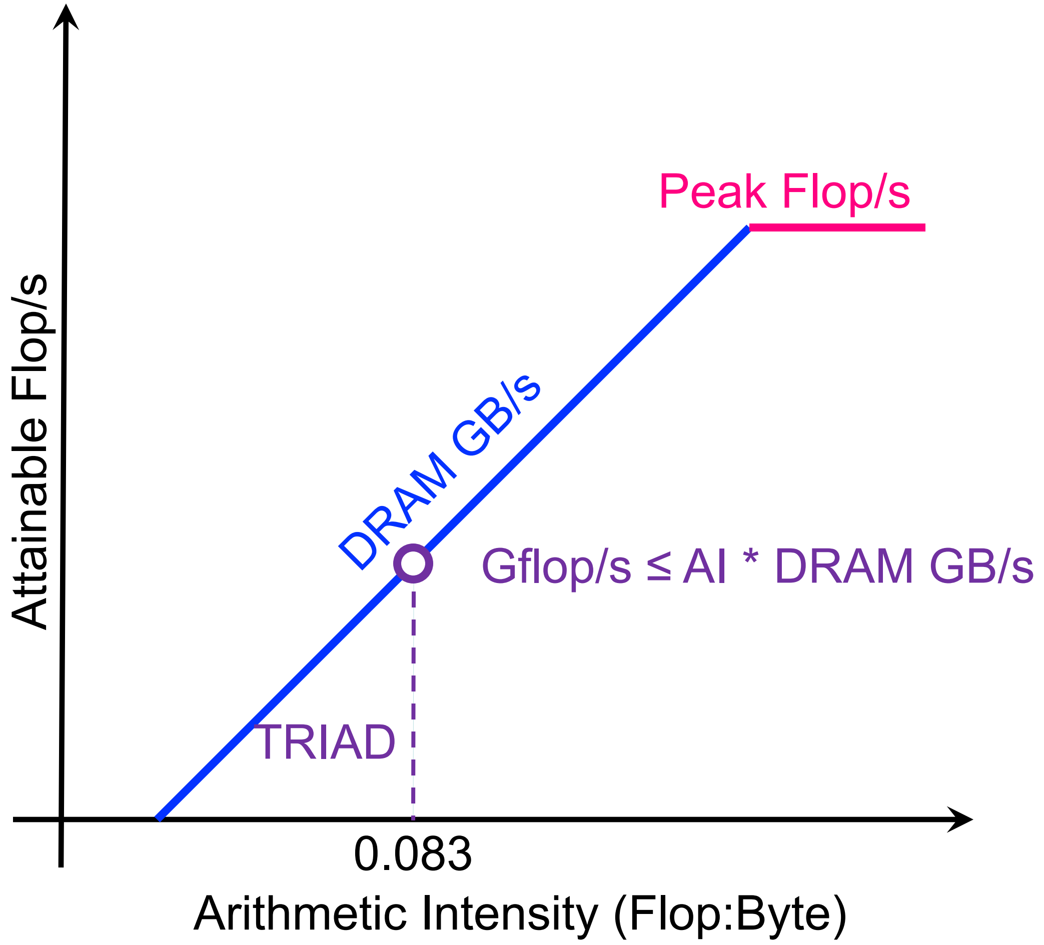
Roofline Example #1

- 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];
}
```

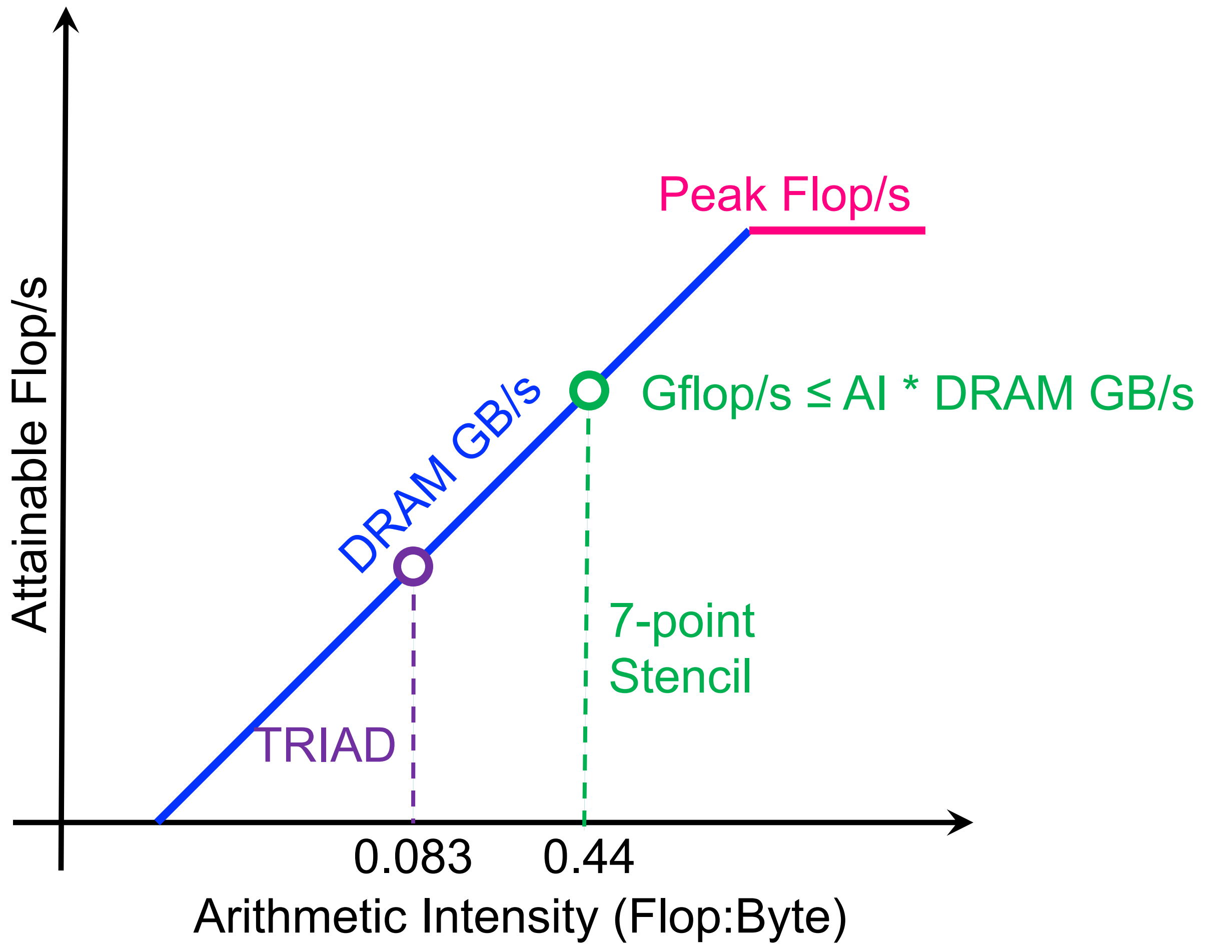
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 flops per byte == Memory bound**



Roofline Example #2

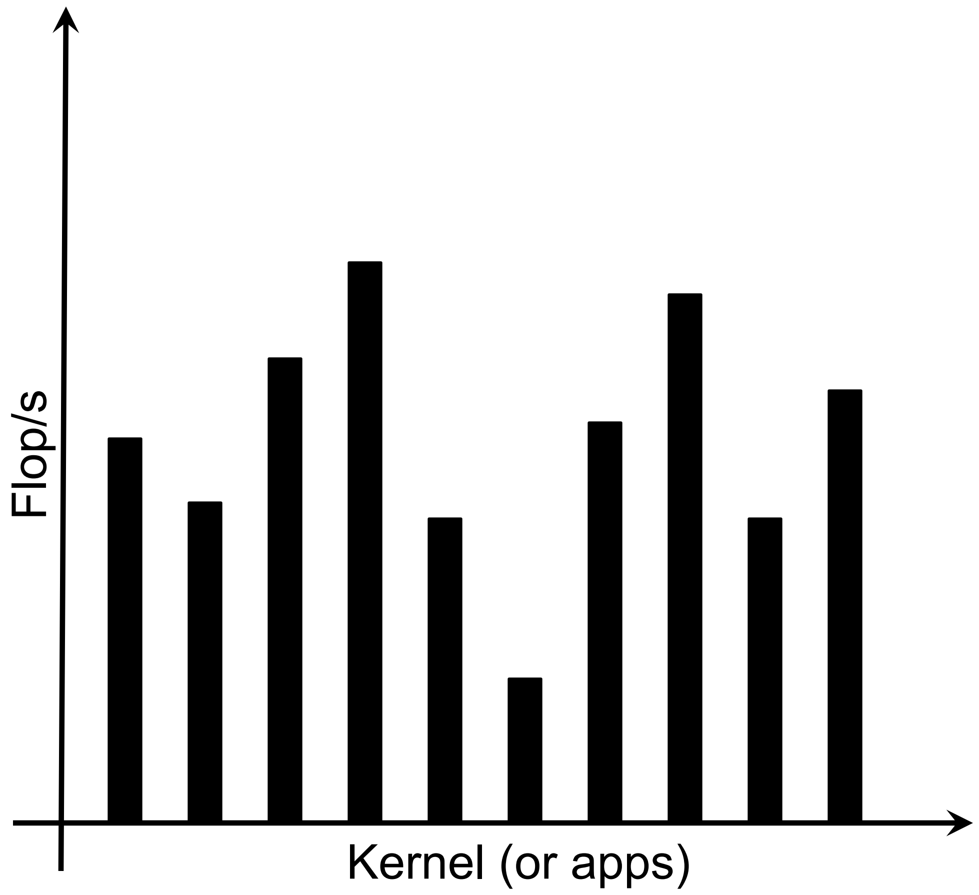
- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - Cache can filter all but 1 read and 1 write per point
  - **AI = 0.44 flops per byte == memory bound, but 5x the flop rate**

```
#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++){
  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  ];
}}}
```



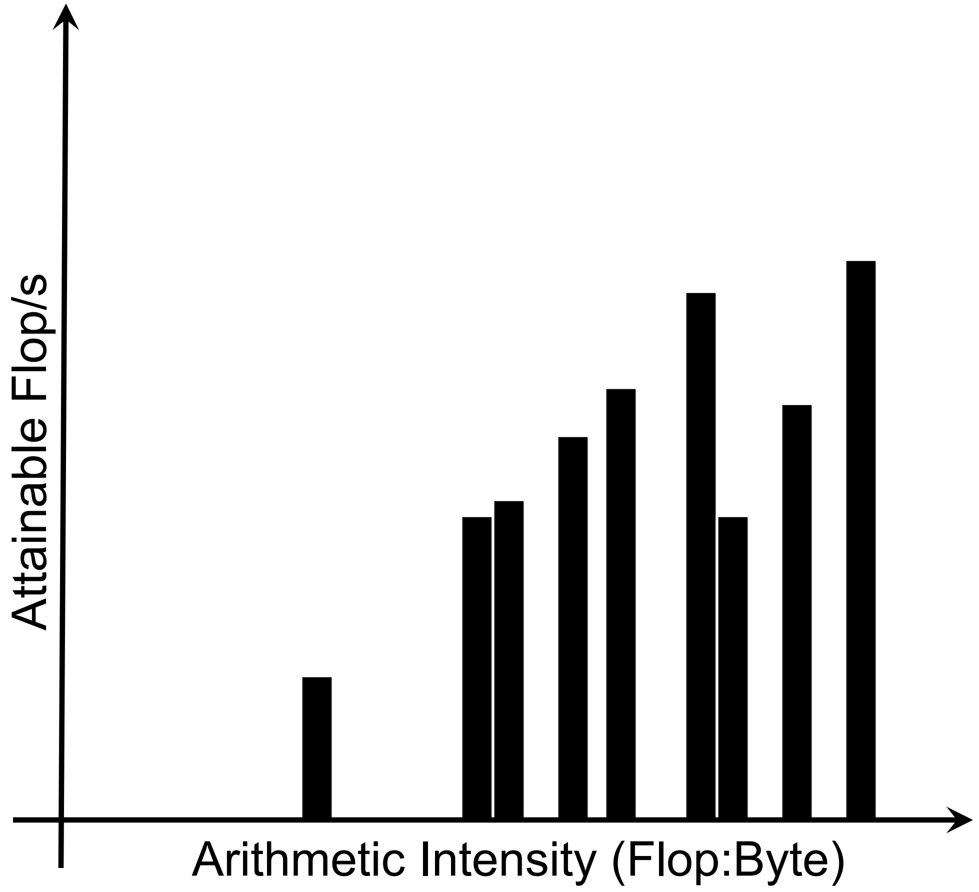
Why is Roofline Useful?

- Imagine a mix of loop nests
- Flop/s alone may not be useful in deciding which to optimize first



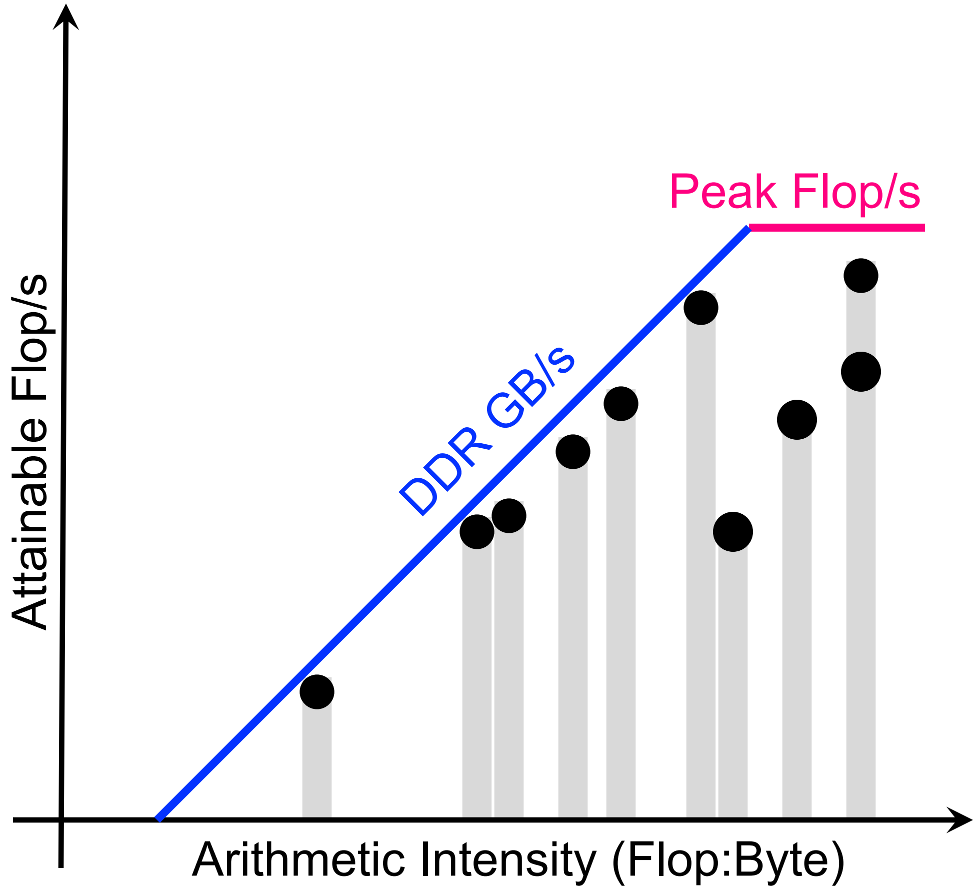
Why is Roofline Useful?

- We can sort kernels by AI …



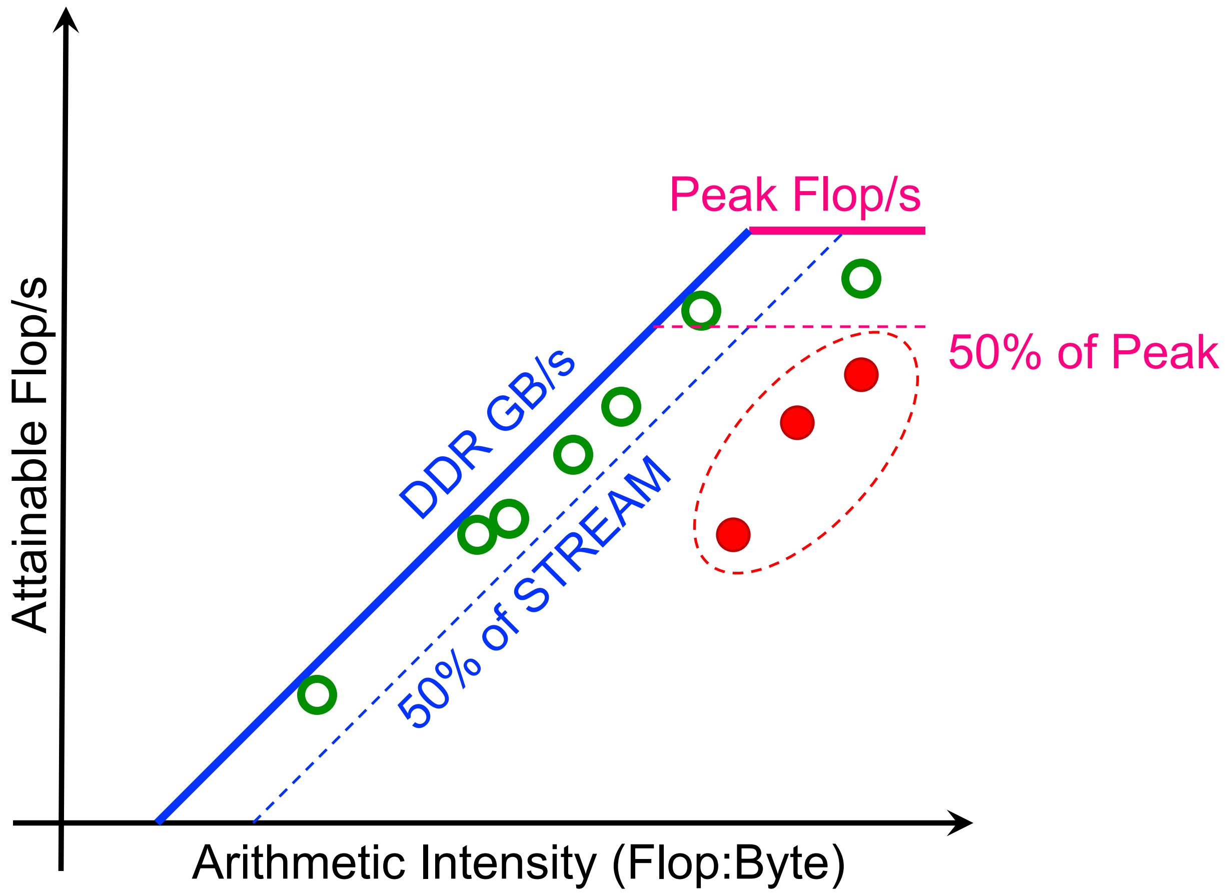
Why is Roofline Useful?

- 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



Refining Roofline:

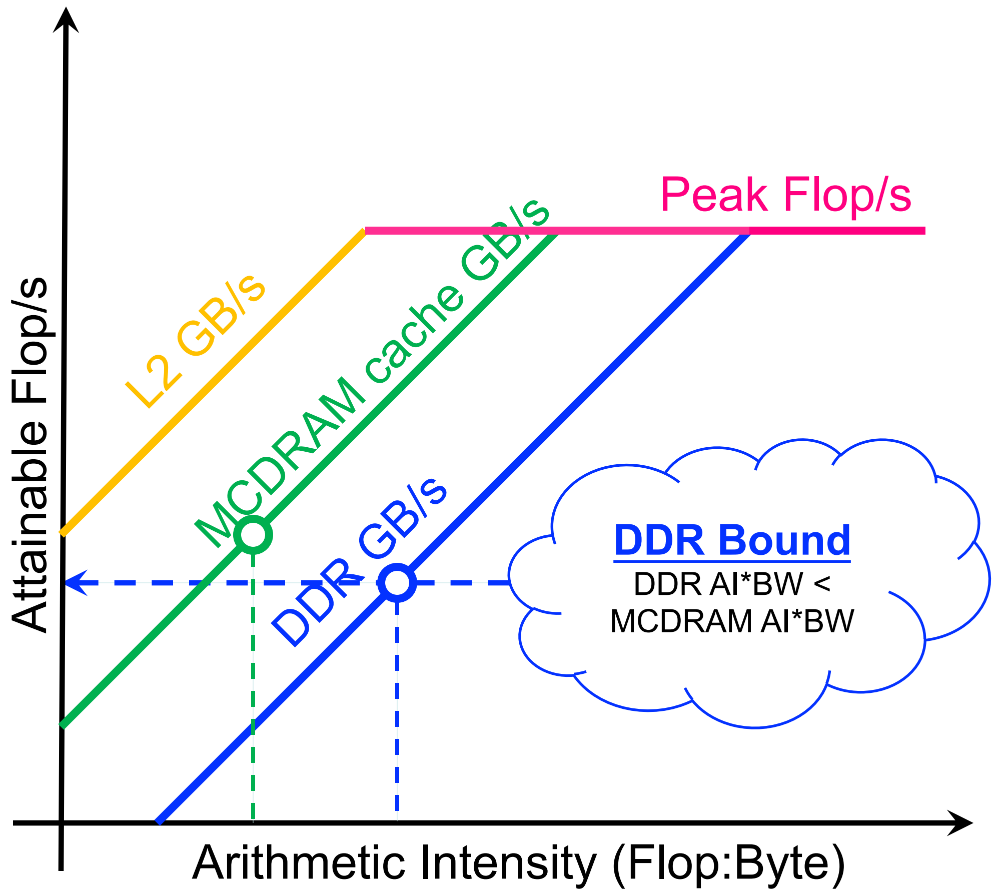
Memory Hierarchy & DLP

Hierarchical Roofline

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

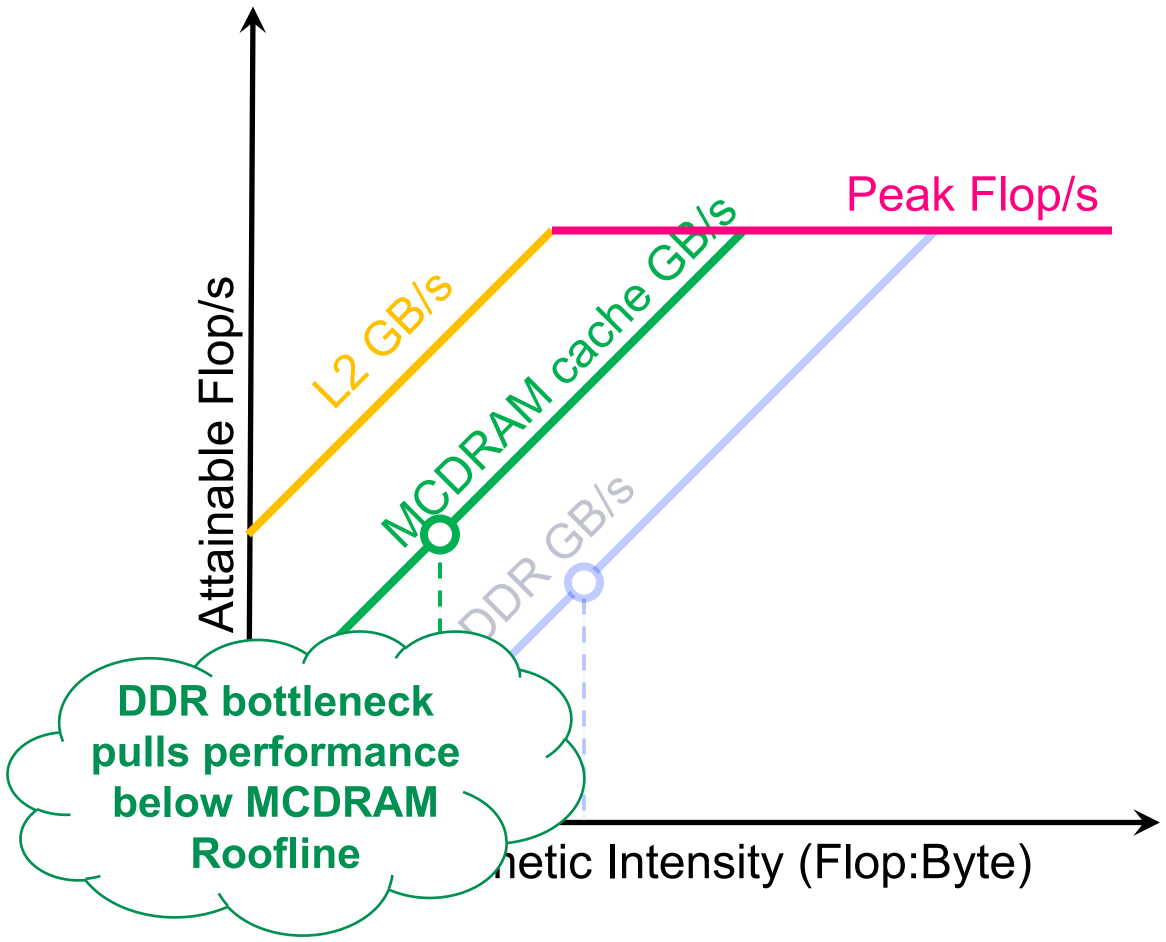
Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure 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

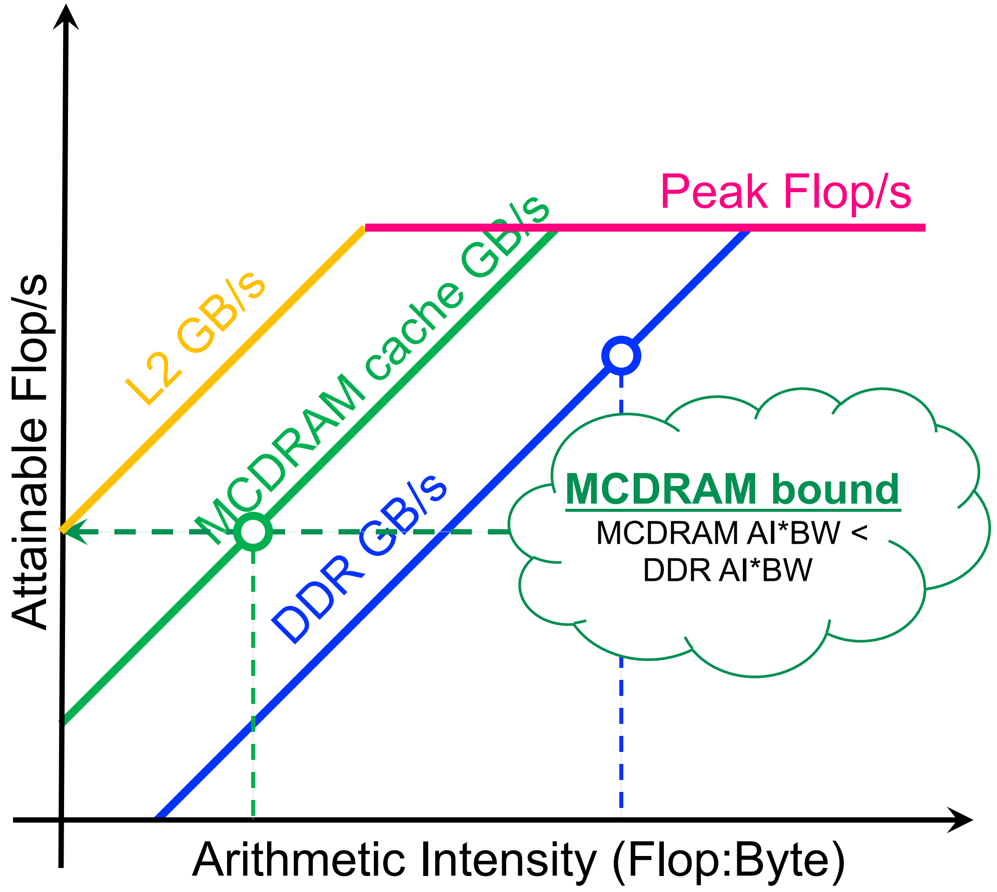
- Construct superposition of Rooflines…
  - Measure bandwidth
  - Measure AI for each level of memory
  - Although an loop nest may have multiple AI’s and multiple bounds (flops, L1, L2, … DRAM)…
  - **… performance is bound by the minimum**



DDR bottleneck pulls performance below MCDRAM Roofline

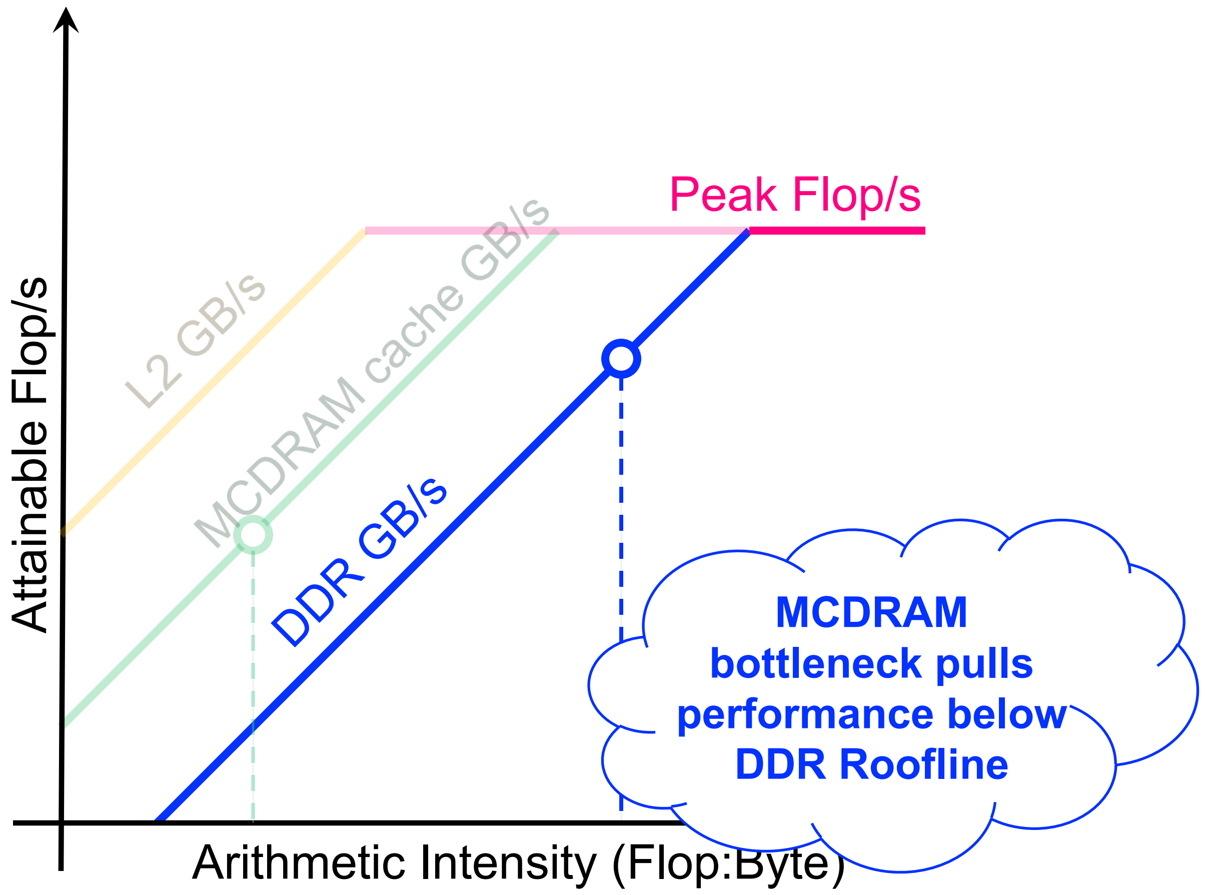
Hierarchical Roofline

- Construct superposition of Rooflines…
  - Measure 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

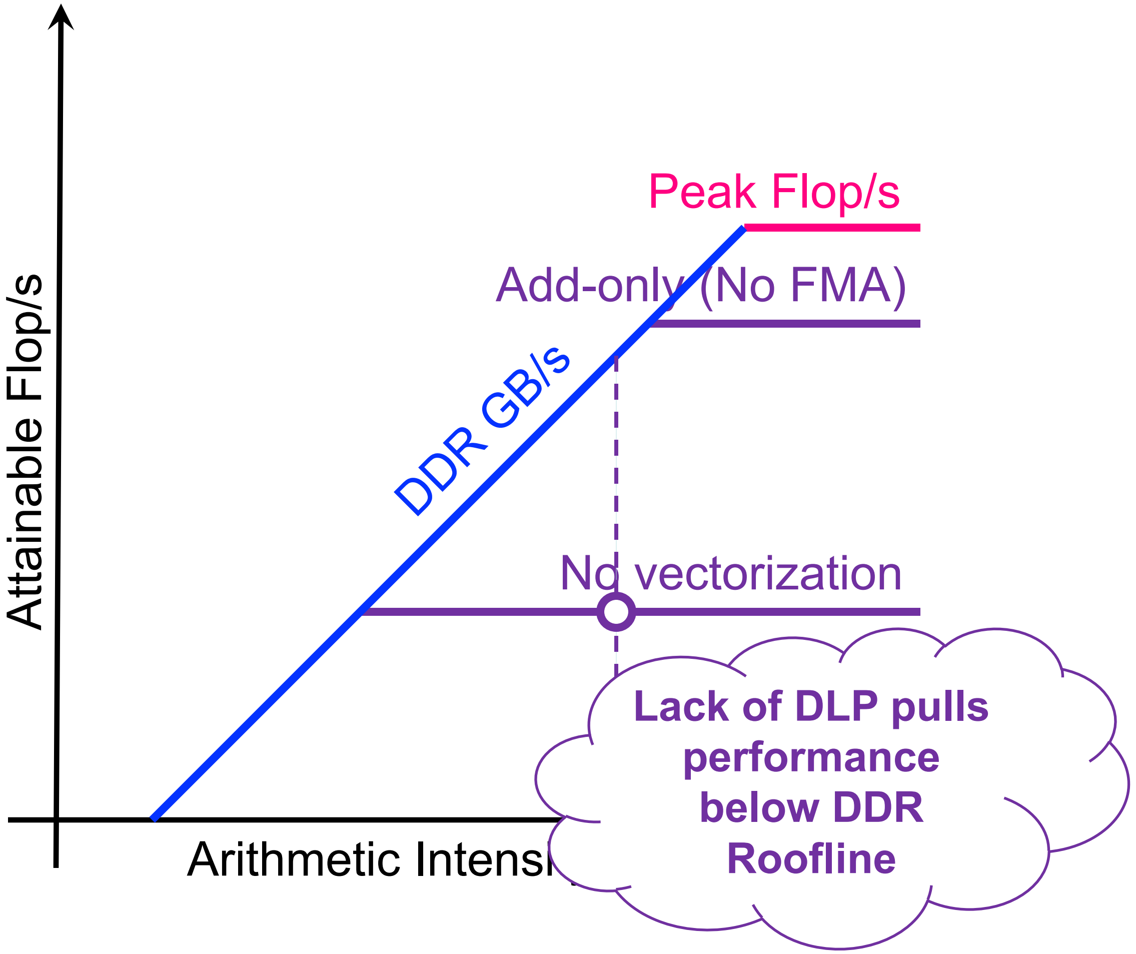
- Construct superposition of Rooflines…
  - Measure bandwidth
  - Measure AI for each level of memory
  - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, … DRAM)…
  - **… performance is bound by the minimum**



MCDRAM bottleneck pulls performance below DDR Roofline

Data, Instruction, Thread-Level Parallelism…

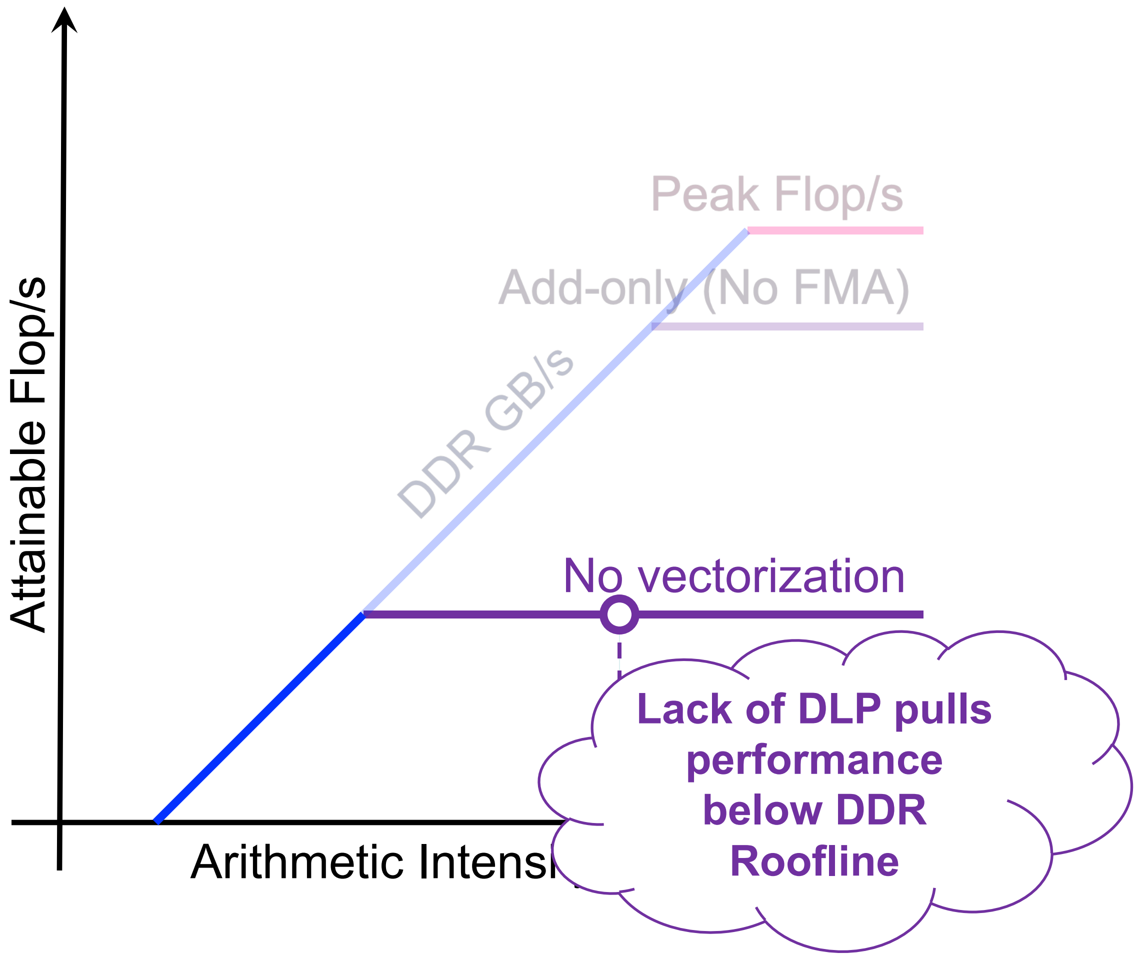
- We have assumed one can attain peak flops with high locality.
- In reality, we must …
  - Use special instructions (e.g. FMA)
  - Vectorize loops (16 flops per instruction)
  - Hide FPU latency (unrolling, out-of-order execution)
  - Use all cores & sockets
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound



Lack of DLP pulls performance below DDR Roofline

Data, Instruction, Thread-Level Parallelism…

- We have assumed one can attain peak flops with high locality.
- In reality, we must …
  - Use special instructions (e.g. FMA)
  - Vectorize loops (16 flops per instruction)
  - Hide FPU latency (unrolling, out-of-order execution)
  - Use all cores & sockets
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound

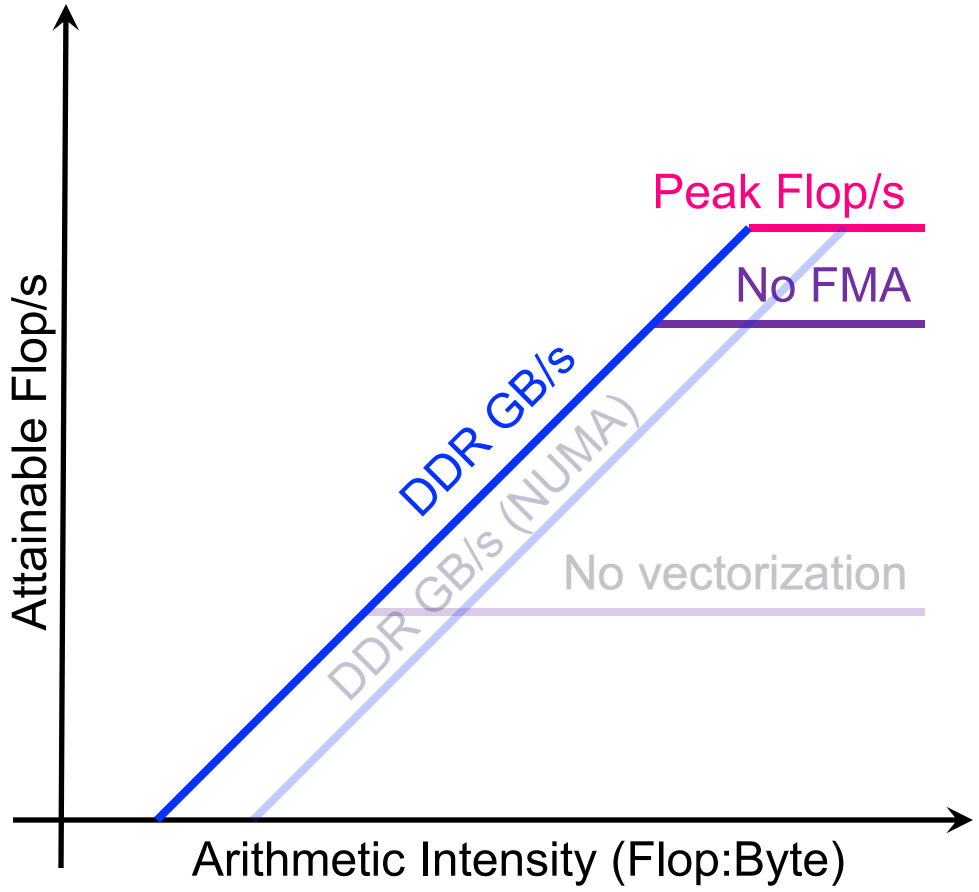


Roofline Model:

Roofline-driven Performance Optimization

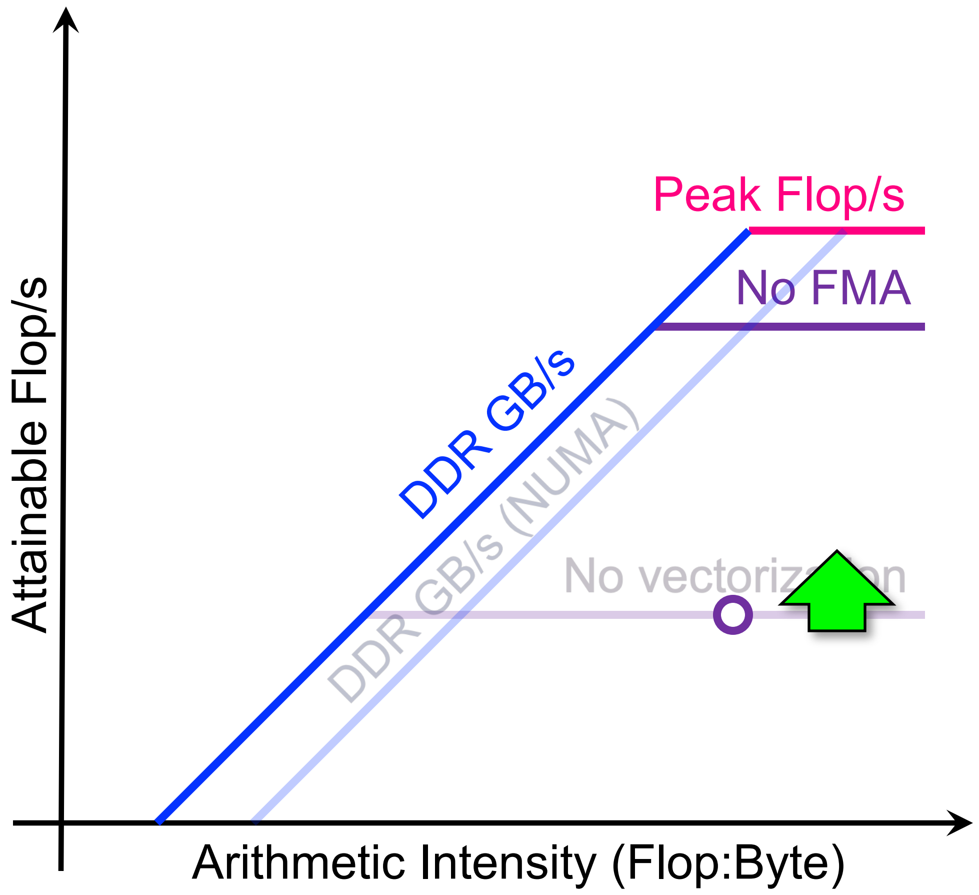
Roofline-Driven Performance Optimization

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



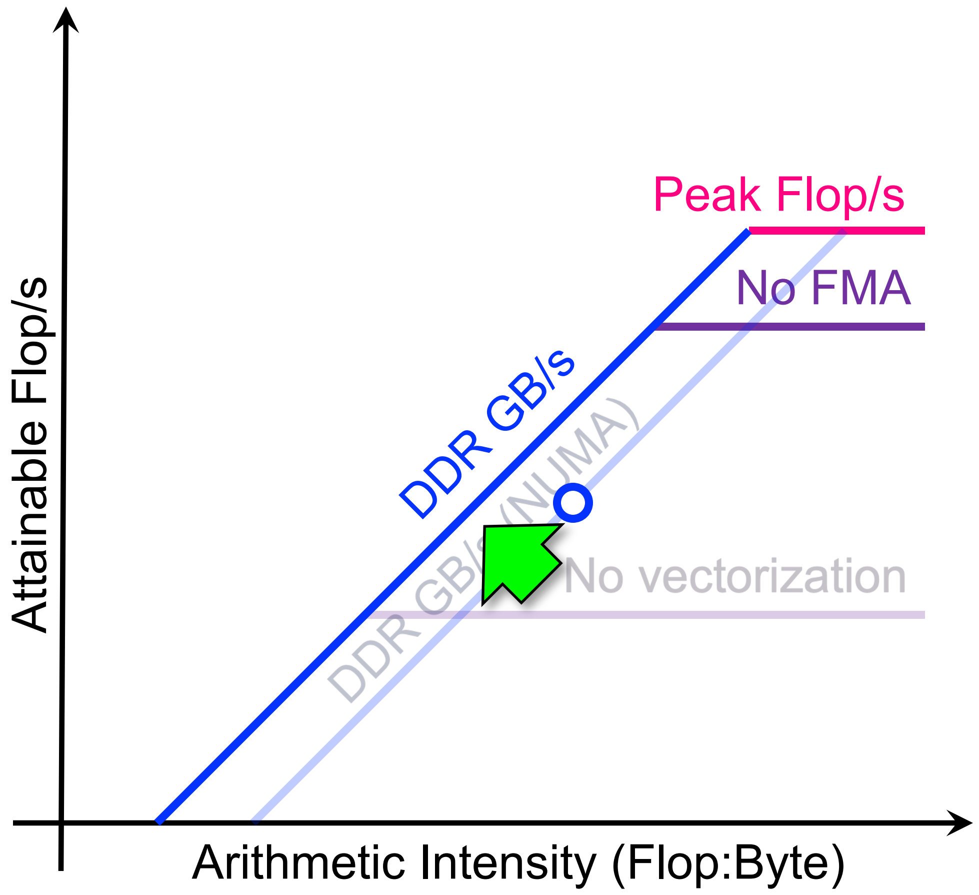
Roofline-Driven Performance Optimization

- Broadly speaking, there are three approaches to improving performance:
- **Maximize in-core performance (e.g. get compiler to vectorize)**



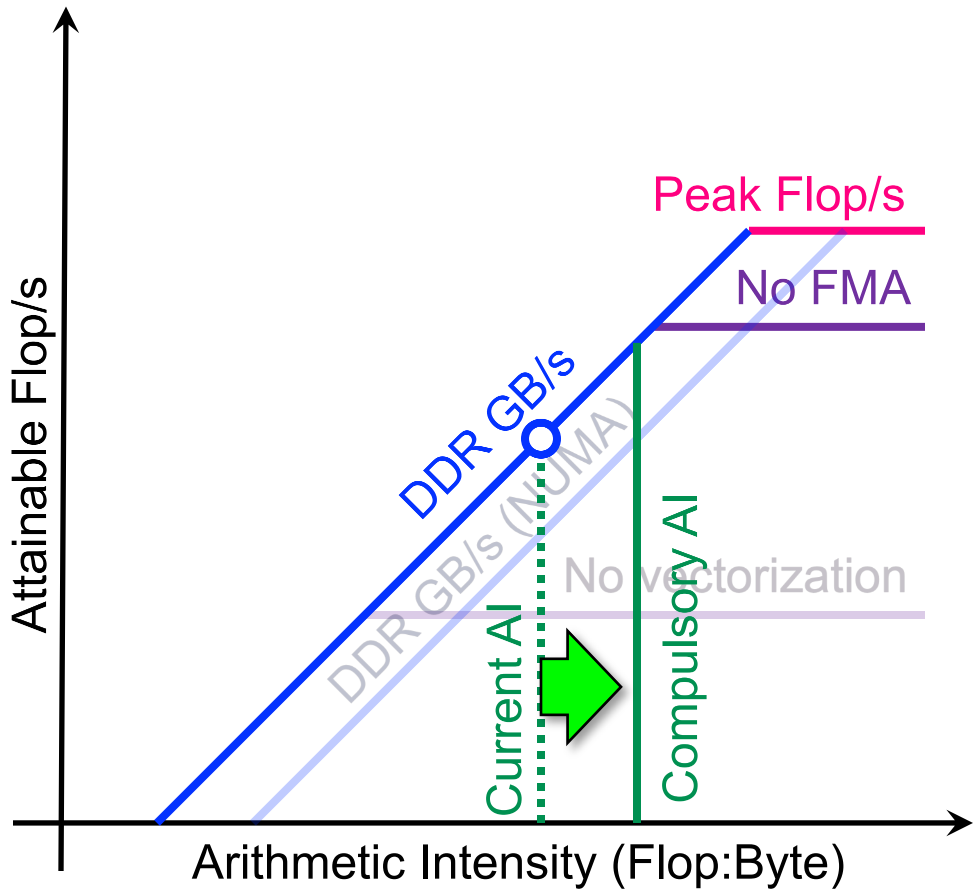
Roofline-Driven Performance Optimization

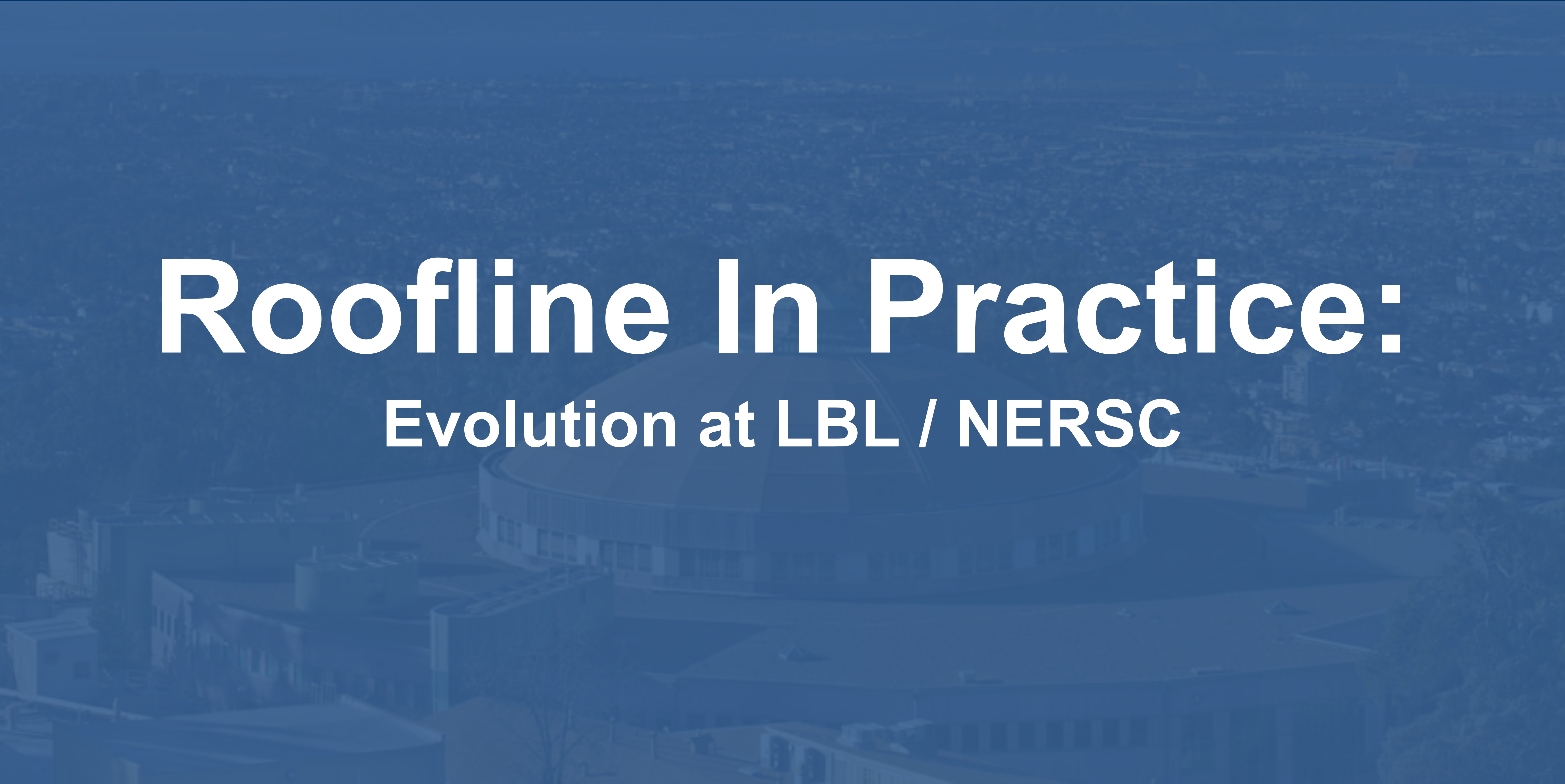
- Broadly speaking, there are three approaches to improving performance:
- Maximize in-core performance (e.g. get compiler to vectorize)
- **Maximize memory bandwidth (e.g. NUMA-aware, unit-stride)**



Roofline-Driven Performance Optimization

- Broadly speaking, there are three approaches to improving performance:
- Maximize in-core performance (e.g. get compiler to vectorize)
- Maximize memory bandwidth (e.g. NUMA-aware, unit stride)
- **Minimize data movement (e.g. cache blocking)**





Roofline In Practice:

Evolution at LBL / NERSC

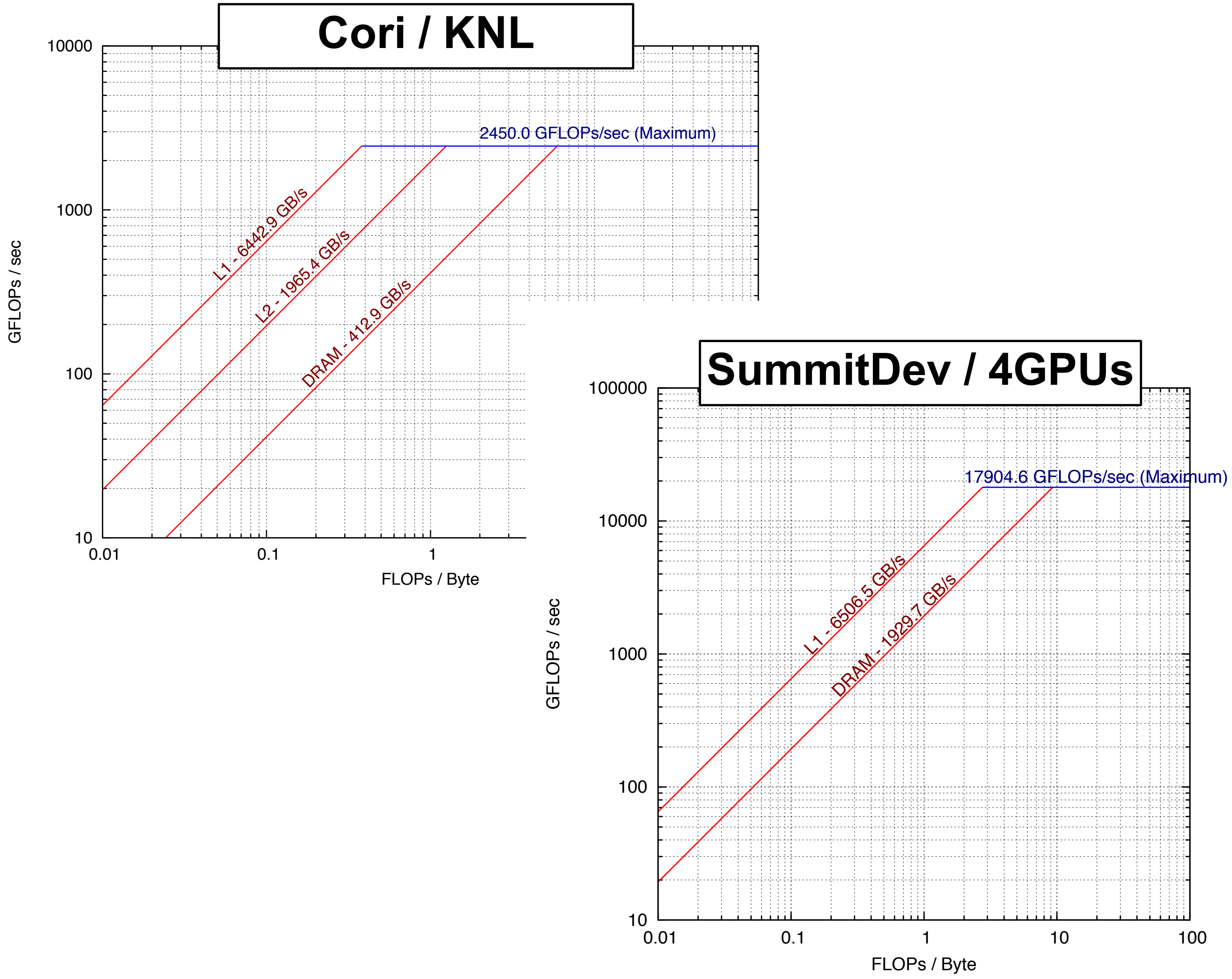
Step 1:

Machine Characterization

Machine Characterization

- **"Theoretical Performance"** numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode / Underclock for AVX
  - 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**



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

Step 2:

Application Characterization

Measuring AI

- To characterize execution with Roofline we need…
  - **Time**
  - **Flops** (=> flop's / time)
  - **Data movement** between each level of memory (=> Flop's / GB's)
- We can look at the full application…
  - Coarse grained, 30-min average
  - Misses many details and bottlenecks
- or we can look at individual loop nests…
  - Requires auto-instrumentation on a loop by loop basis
  - Moreover, we should probably differentiate data movement or flops on a core-by-core basis.

How Do We Count Flop's?

Manual Counting

- ▪ Go thru each loop nest and count the number of FP operations
- ✓ Works best for deterministic loop bounds
- ✓ or parameterize by the number of iterations (recorded at run time)
- ✗ Not scalable

Perf. Counters

- ▪ Read counter before/after
- ✓ More Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✗ Requires privileged access
- ✗ Requires manual instrumentation (+overhead) or full-app characterization
- ✗ Broken counters = garbage
- ✗ May not differentiate FMADD from FADD
- ✗ No insight into special pipelines

Binary Instrumentation

- ▪ Automated inspection of assembly at run time
- ✓ Most Accurate
- ✓ FMA-, VL-, and mask-aware
- ✓ Can count instructions by class/type
- ✓ Can detect load imbalance
- ✓ Can include effects from non-FP instructions
- ✓ Automated application to multiple loop nests
- ✗ >10x overhead (short runs / reduced concurrency)

How Do We Measure Data Movement?

Manual Counting

- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches

✓ Works best for simple loops that stream from DRAM (stencils, FFTs, spare, …)

✗ N/A for complex caches

✗ Not scalable

Perf. Counters

- Read counter before/after

✓ Applies to full hierarchy (L2, DRAM,

✓ Much more Accurate

✓ Low overhead (<%) == can run full MPI applications

✓ Can detect load imbalance

✗ Requires privileged access

✗ Requires manual instrumentation (+overhead) or full-app characterization

Cache Simulation

- Build a full cache simulator driven by memory addresses

✓ Applies to full hierarchy and multicore

✓ Can detect load imbalance

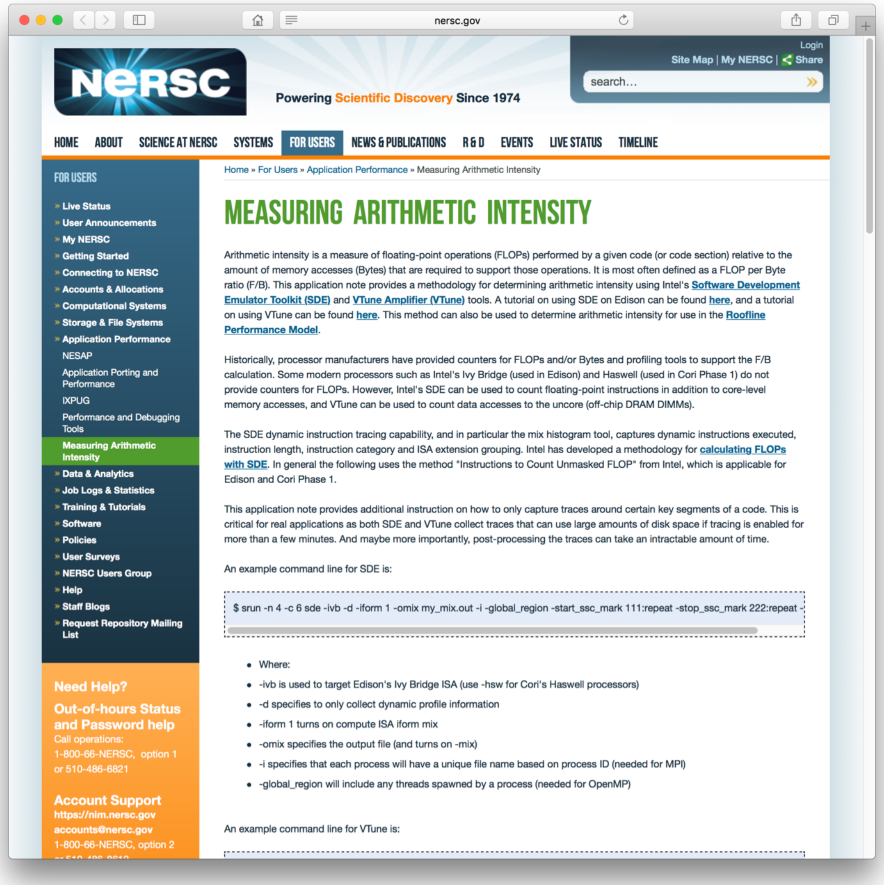
✓ Automated application to multiple loop nests

✗ Ignores prefetchers

✗ >10x overhead (short runs / reduced concurrency)

Previously Cobbled 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…



Home » For Users » Application Performance » Measuring Arithmetic Intensity

MEASURING ARITHMETIC INTENSITY

Arithmetic intensity is a measure of floating-point operations (FLOPs) performed by a given code (or code section) relative to the amount of memory accesses (Bytes) that are required to support those operations. It is most often defined as a FLOP per Byte ratio (F/B). This application note provides a methodology for determining arithmetic intensity using Intel's Software Development Emulator Toolkit (SDE) and VTune Amplifier (VTune) tools. A tutorial on using SDE on Edison can be found here, and a tutorial on using VTune can be found here. This method can also be used to determine arithmetic intensity for use in the Roofline Performance Model.

Historically, processor manufacturers have provided counters for FLOPs and/or Bytes and profiling tools to support the F/B calculation. Some modern processors such as Intel's Ivy Bridge (used in Edison) and Haswell (used in Cori Phase 1) do not provide counters for FLOPs. However, Intel's SDE can be used to count floating-point instructions in addition to core-level memory accesses, and VTune can be used to count data accesses to the uncore (off-chip DRAM DIMMs).

The SDE dynamic instruction tracing capability, and in particular the mix histogram tool, captures dynamic instructions executed, instruction length, instruction category and ISA extension grouping. Intel has developed a methodology for calculating FLOPs with SDE. In general the following uses the method "Instructions to Count Unmasked FLOP" from Intel, which is applicable for Edison and Cori Phase 1.

This application note provides additional instruction on how to only capture traces around certain key segments of a code. This is critical for real applications as both SDE and VTune collect traces that can use large amounts of disk space if tracing is enabled for more than a few minutes. And maybe more importantly, post-processing the traces can take an intractable amount of time.

An example command line for SDE is:

```
$ srun -n 4 -c 6 sde -ivb -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -
```

- Where:
- -ivb is used to target Edison's Ivy Bridge ISA (use -hsw for Cori's Haswell processors)
- -d specifies to only collect dynamic profile information
- -iform 1 turns on compute ISA iform mix
- -omix specifies the output file (and turns on -mix)
- -i specifies that each process will have a unique file name based on process ID (needed for MPI)
- -global_region will include any threads spawned by a process (needed for OpenMP)

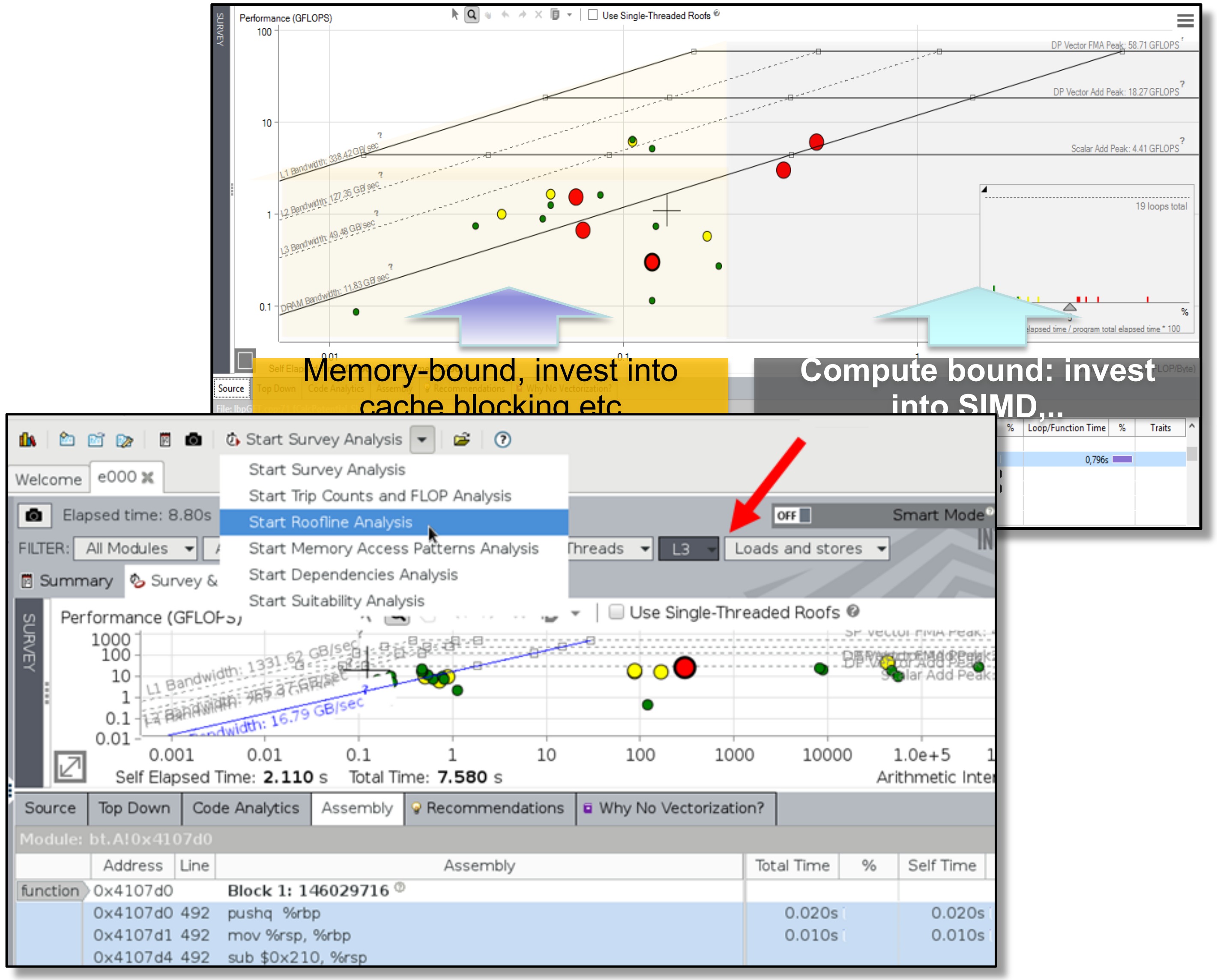
An example command line for VTune is:

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

NERSC is LBL's production computing division
CRD is LBL's Computational Research Division
NESAP is NERSC's KNL application readiness project
LBL is part of SUPER (DOE SciDAC3 Computer Science Institute)

Intel Advisor

- Includes Roofline Automation…
  - ✓ Automatically instruments applications (one dot per loop nest/function)
  - ✓ Computes FLOPS and AI for each function (**CARM**)
  - ✓ AVX-512 support that incorporates masks
  - ✓ **Integrated Cache Simulator[1] (hierarchical roofline / multiple AI’s)**
  - ✓ Automatically benchmarks target system (calculates ceilings)
  - ✓ Full integration with existing Advisor capabilities



Memory-bound, invest into cache blocking etc

Compute bound: invest into SIMD,..

Performance (GFLOPS)
Use Single-Threaded Roofs
DP Vector FMA Peak: 58.71 GFLOPS
DP Vector Add Peak: 18.27 GFLOPS
Scalar Add Peak: 4.41 GFLOPS
L1 Bandwidth: 308.42 GB/sec
L2 Bandwidth: 127.35 GB/sec
L3 Bandwidth: 49.48 GB/sec
DRAM Bandwidth: 11.83 GB/sec
19 loops total
Start Survey Analysis
Welcome
e000
Start Survey Analysis
Start Trip Counts and FLOP Analysis
Start Roofline Analysis
Start Memory Access Patterns Analysis
Start Dependencies Analysis
Start Suitability Analysis
Elapsed time: 8.80s
OFF
Smart Mode
FILTER: All Modules
Threads
L3
Loads and stores
Summary
Survey &
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
1000
100
10
1
0.1
0.01
L1 Bandwidth: 1331.62 GB/sec
L3 Bandwidth: 465.47 GB/sec
Bandwidth: 16.79 GB/sec
0.001
0.01
0.1
1
10
100
1000
10000
1.0e+5
Arithmetic Inte
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

Self Elapsed Time: **2.110** s   Total Time: **7.580** s

Module: bt.A!0x4107d0

	Address	Line	Assembly	Total Time	%	Self Time
function	0x4107d0		**Block 1: 146029716**			
	0x4107d0	492	pushq %rbp	0.020s		0.020s
	0x4107d1	492	mov %rsp, %rbp	0.010s		0.010s
	0x4107d4	492	sub $0x210, %rsp			

http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017

[1] Experimental Feature, the look and feel and exact behavior is subject for change

Hierarchical Roofline vs. Cache-Aware Roofline

…understanding different Roofline formulations in Intel 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 looses 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 Roofline[1] (cache simulator) has also been integrated into Intel Advisor

[1] Experimental Feature, the look and feel and exact behavior is subject for change

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

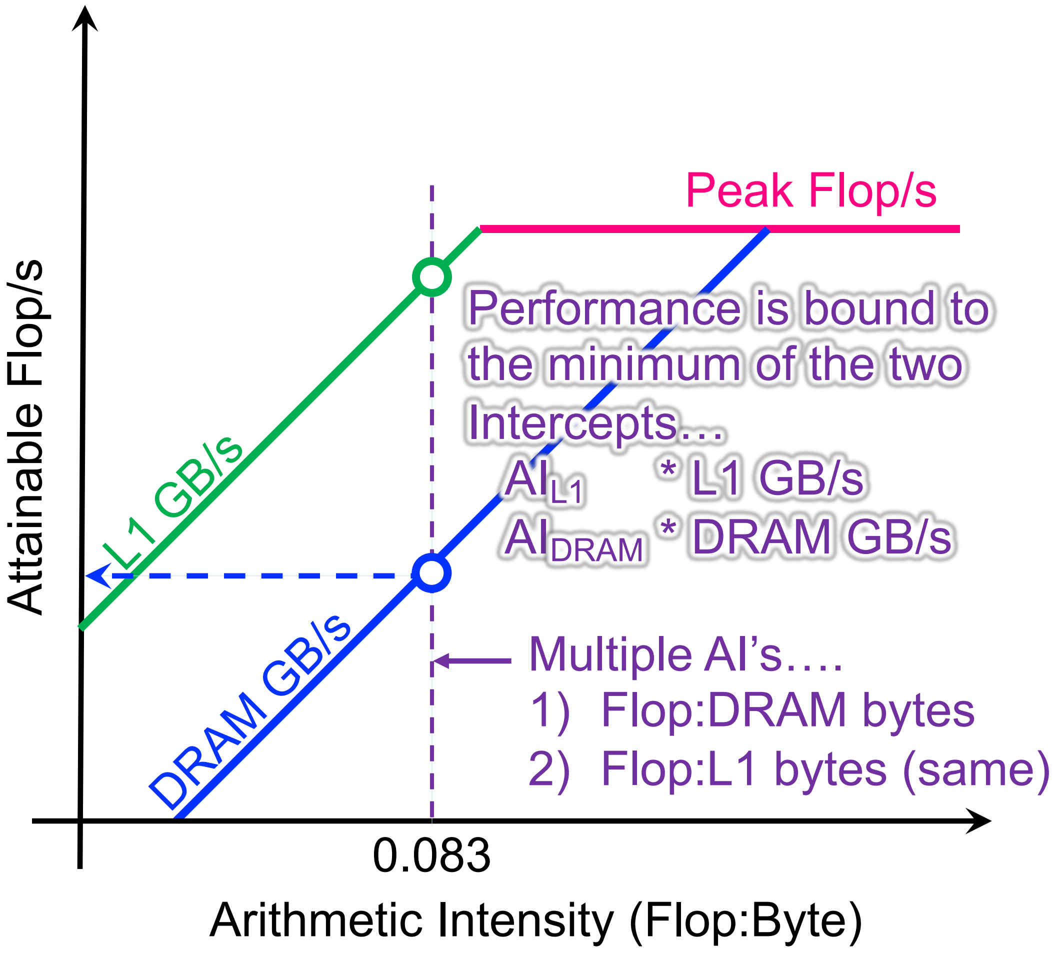
Example: STREAM

- 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+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



Performance is bound to the minimum of the two Intercepts…

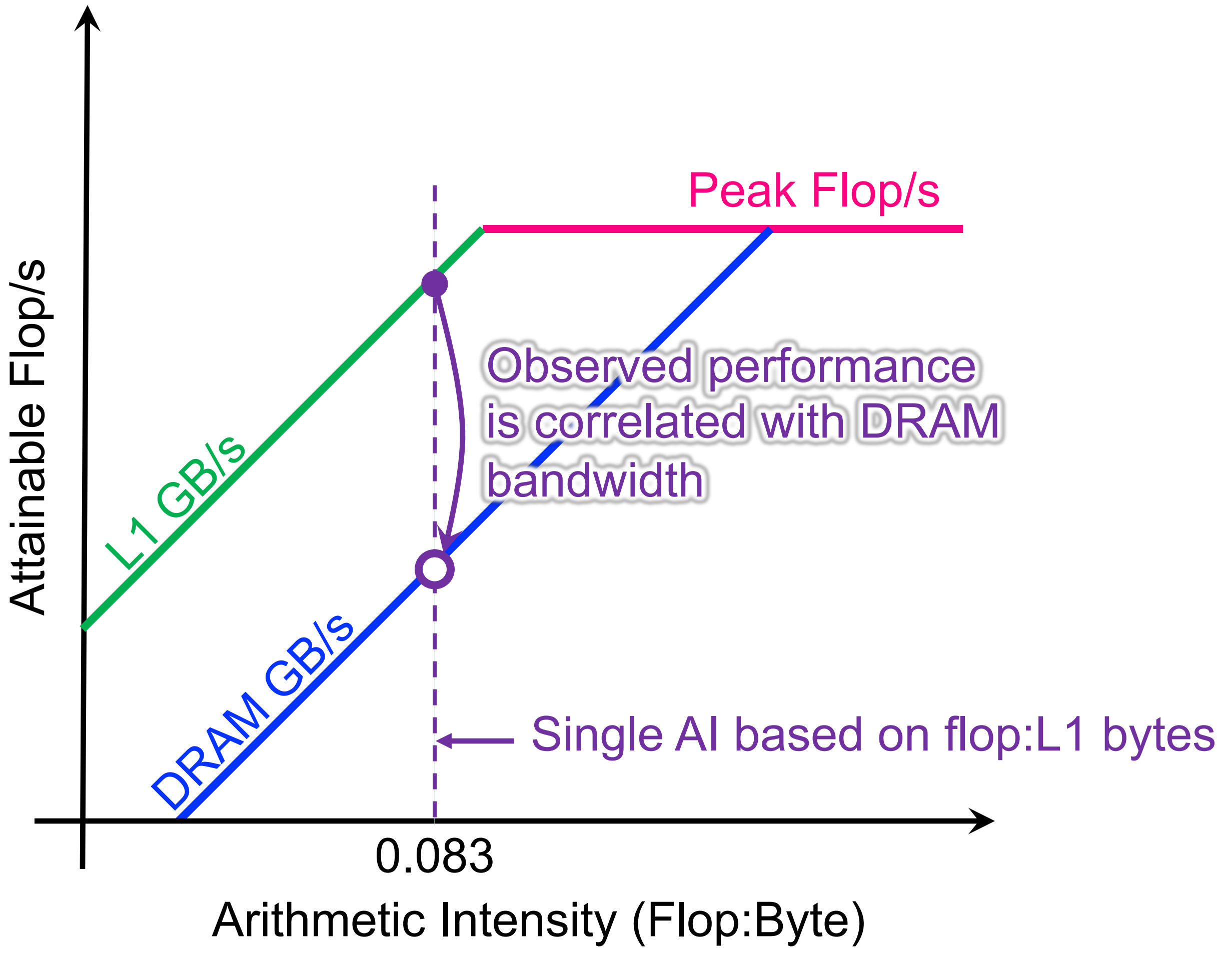
AI_{L1} * L1 GB/s

AI_{DRAM} * DRAM GB/s

Multiple AI's….

1) Flop:DRAM bytes
2) Flop:L1 bytes (same)

Cache-Aware Roofline



Observed performance is correlated with DRAM bandwidth

Single AI based on flop:L1 bytes

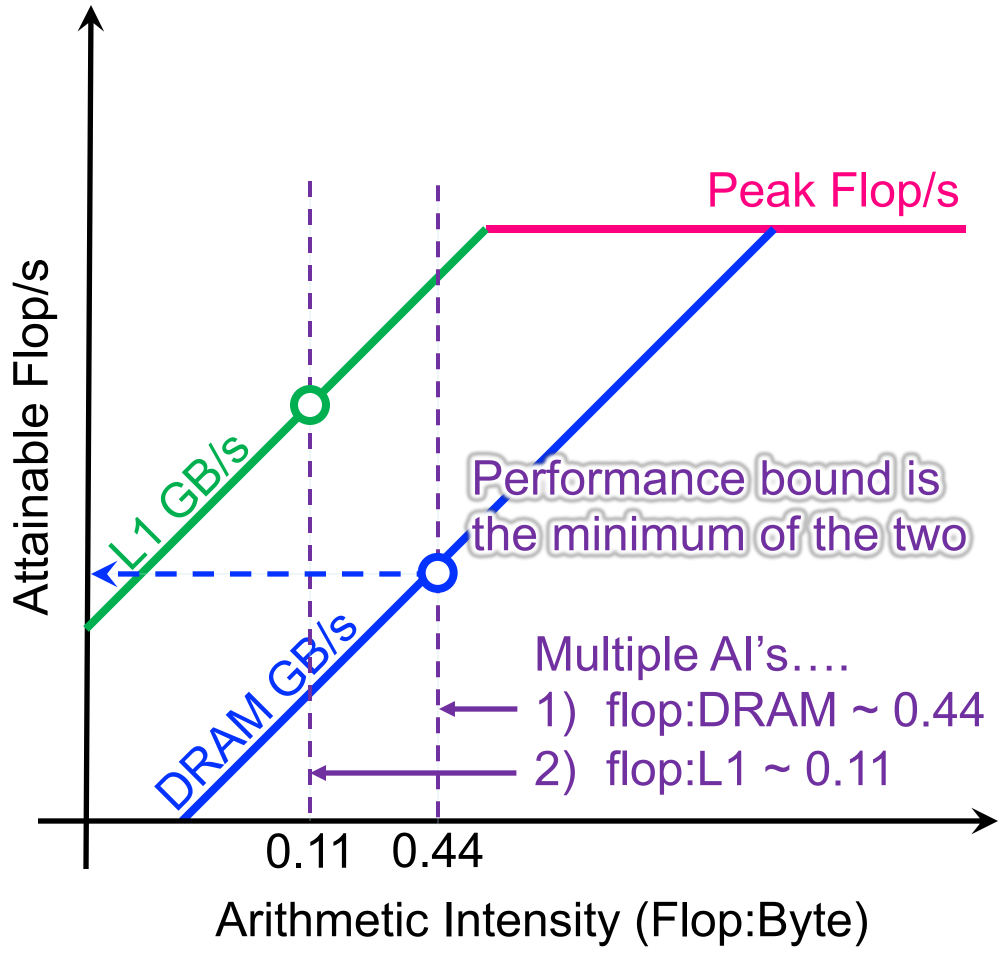
Example: 7-point Stencil (Small Problem)

- 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[k][j][i+1]` is reused on next iteration of i.
  - `old[k][j+1][i]` is reused on next iteration of j.
  - `old[k+1][j][i]` is reused on next iterations of k.
- … leads to DRAM AI larger than the L1 AI

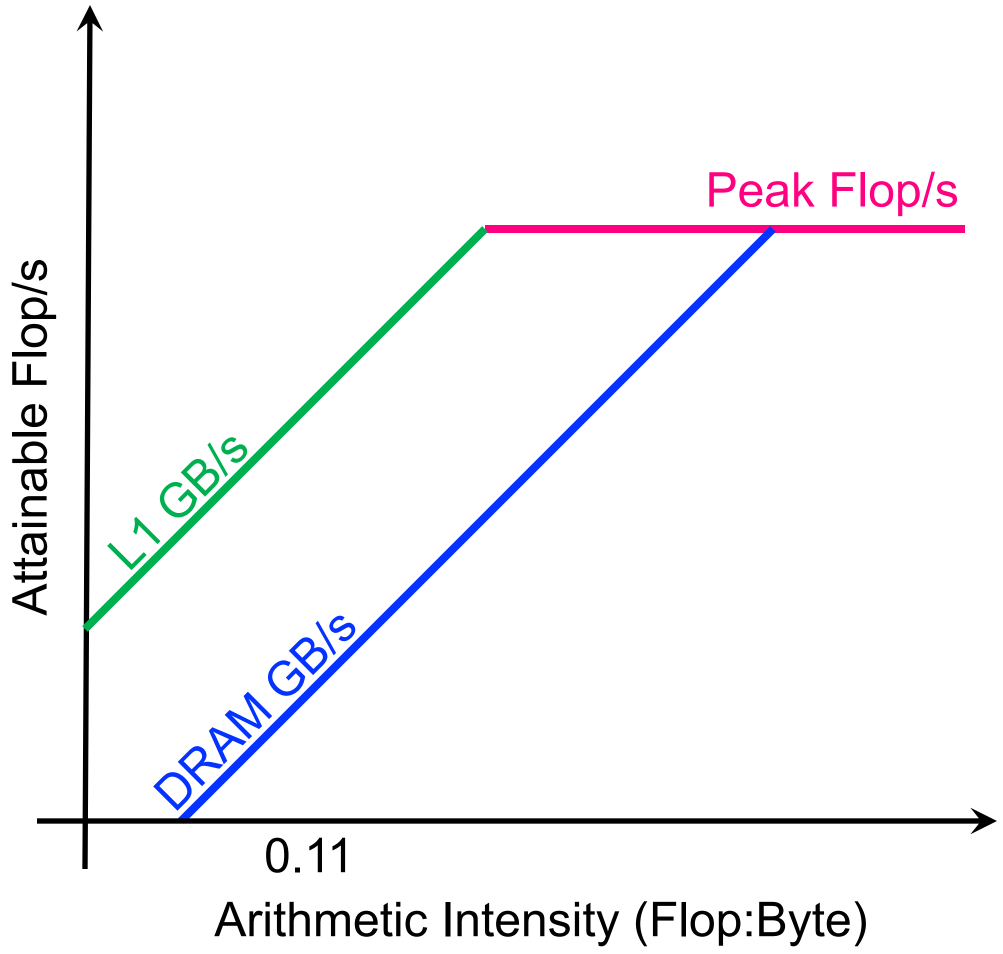
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                    + old[k  ][j  ][i-1]
                    + old[k  ][j  ][i+1]
                    + old[k  ][j-1][i  ]
                    + old[k  ][j+1][i  ]
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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

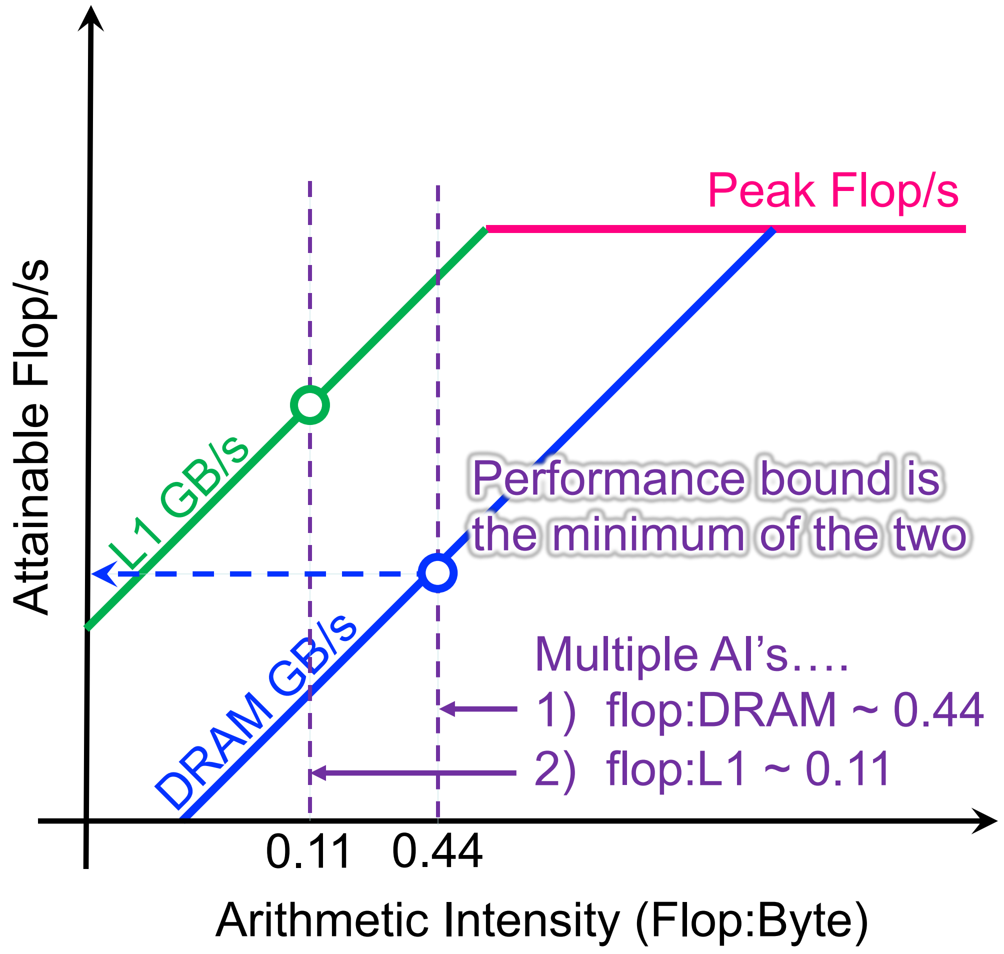


Cache-Aware Roofline



Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

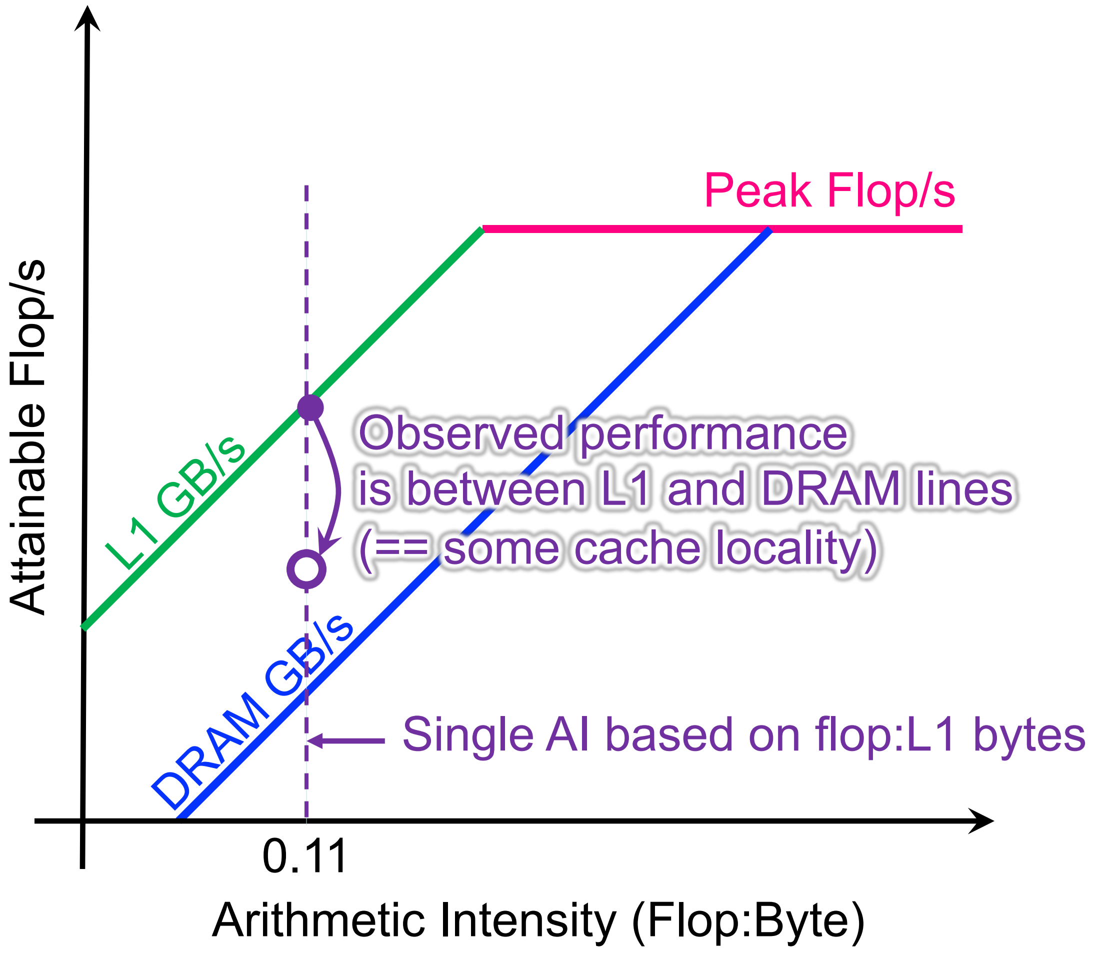


Performance bound is the minimum of the two

Multiple AI's….

1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Cache-Aware Roofline

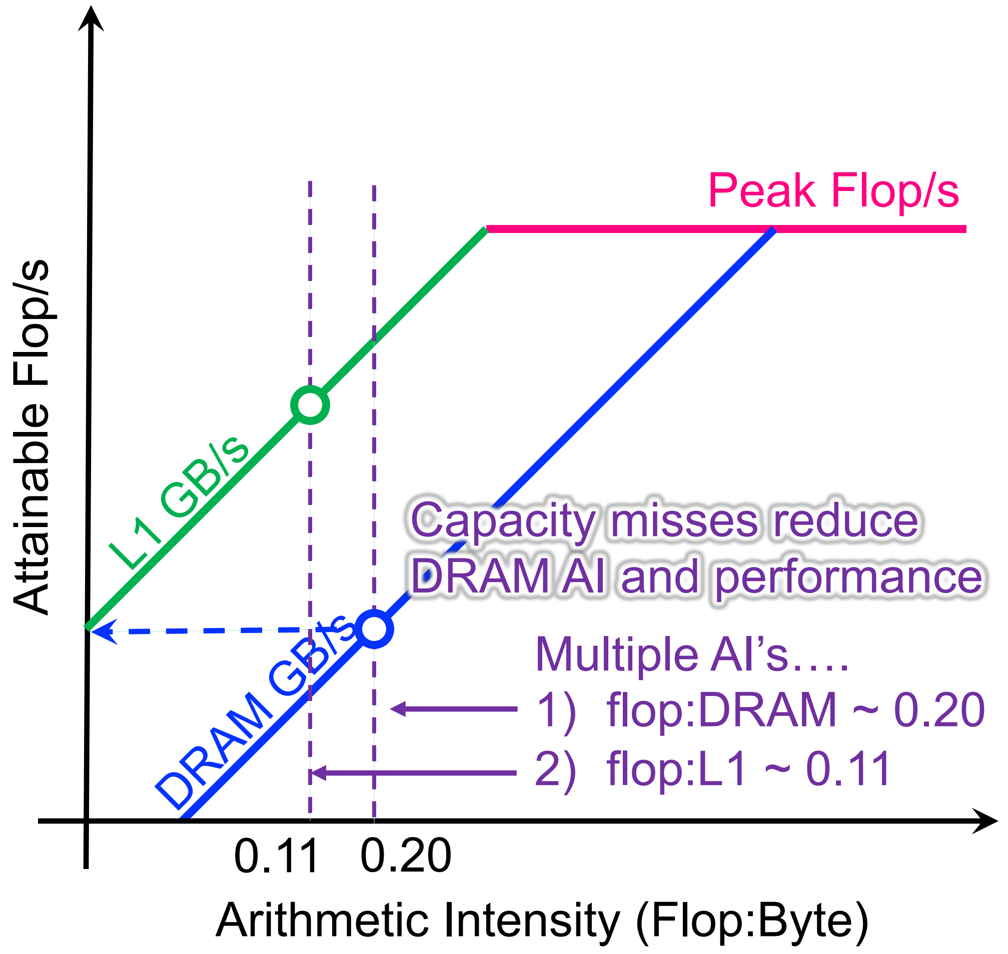


Observed performance is between L1 and DRAM lines (== some cache locality)

Single AI based on flop:L1 bytes

Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

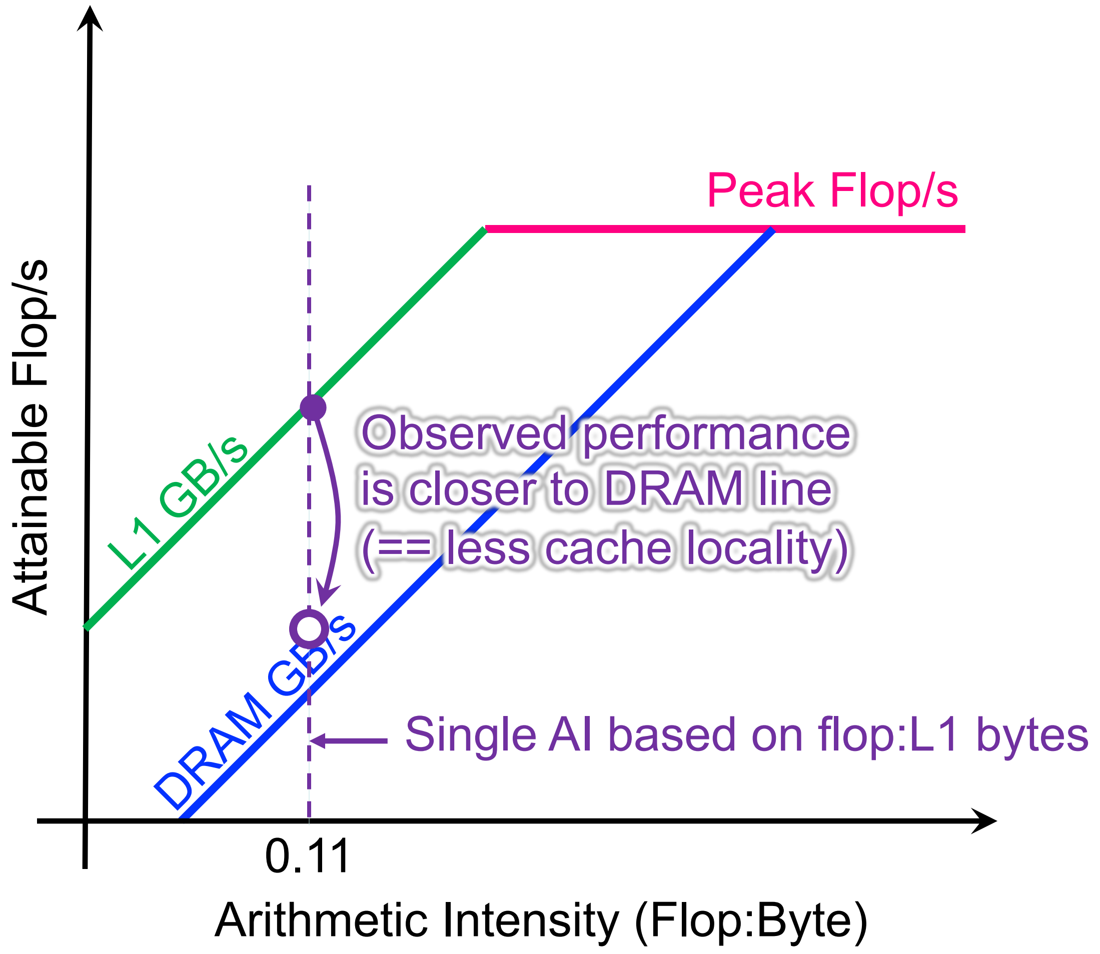


Capacity misses reduce DRAM AI and performance

Multiple AI's….

1) flop:DRAM ~ 0.20
2) flop:L1 ~ 0.11

Cache-Aware Roofline

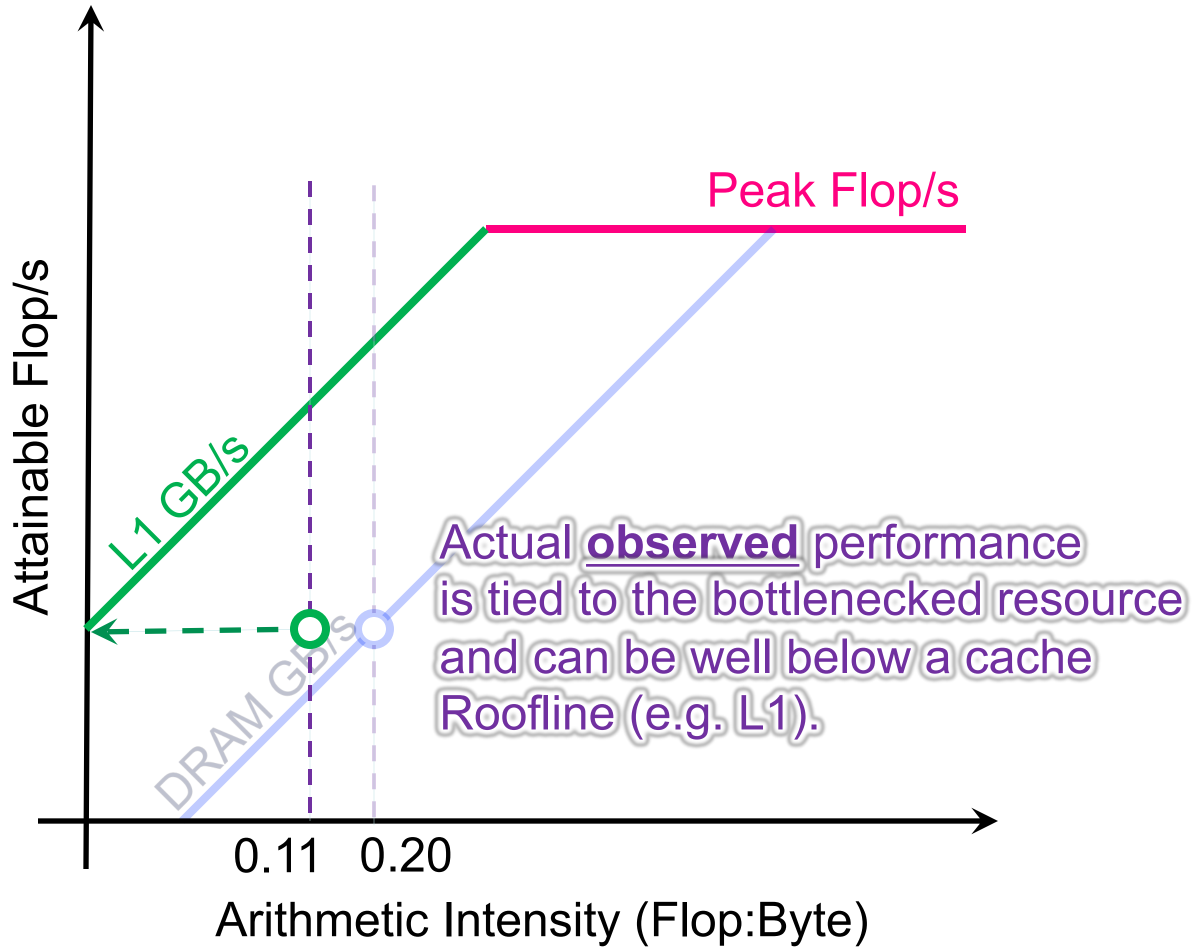


Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes

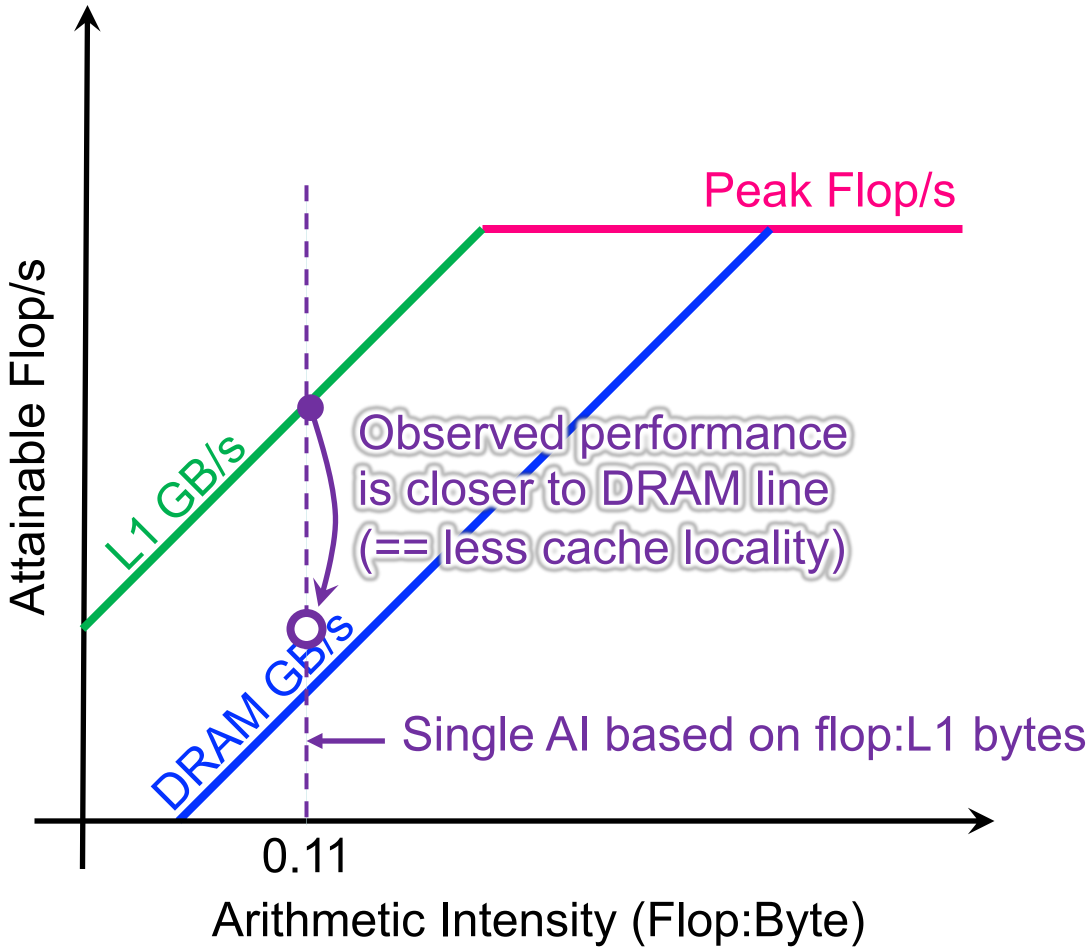
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Actual **observed** performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline

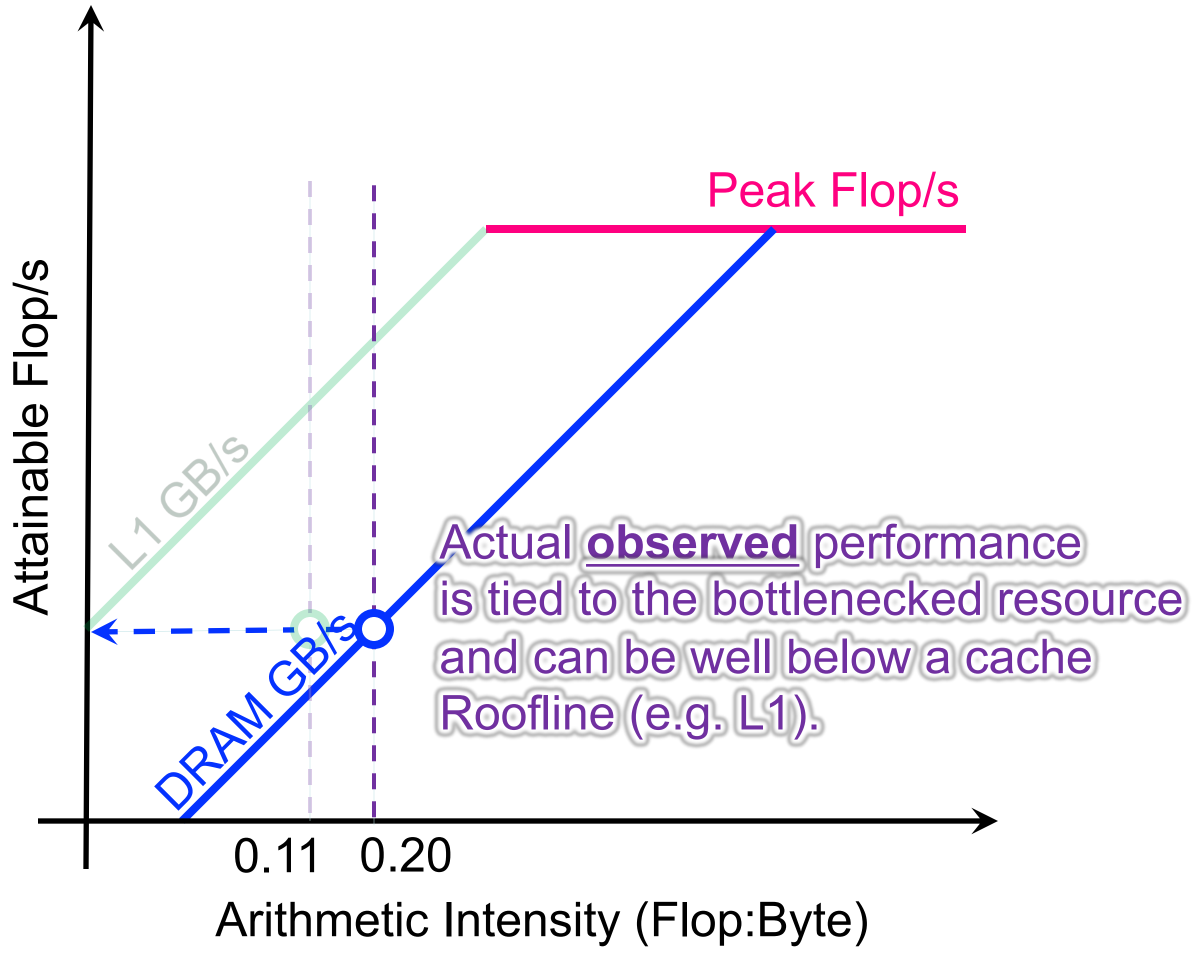


Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes

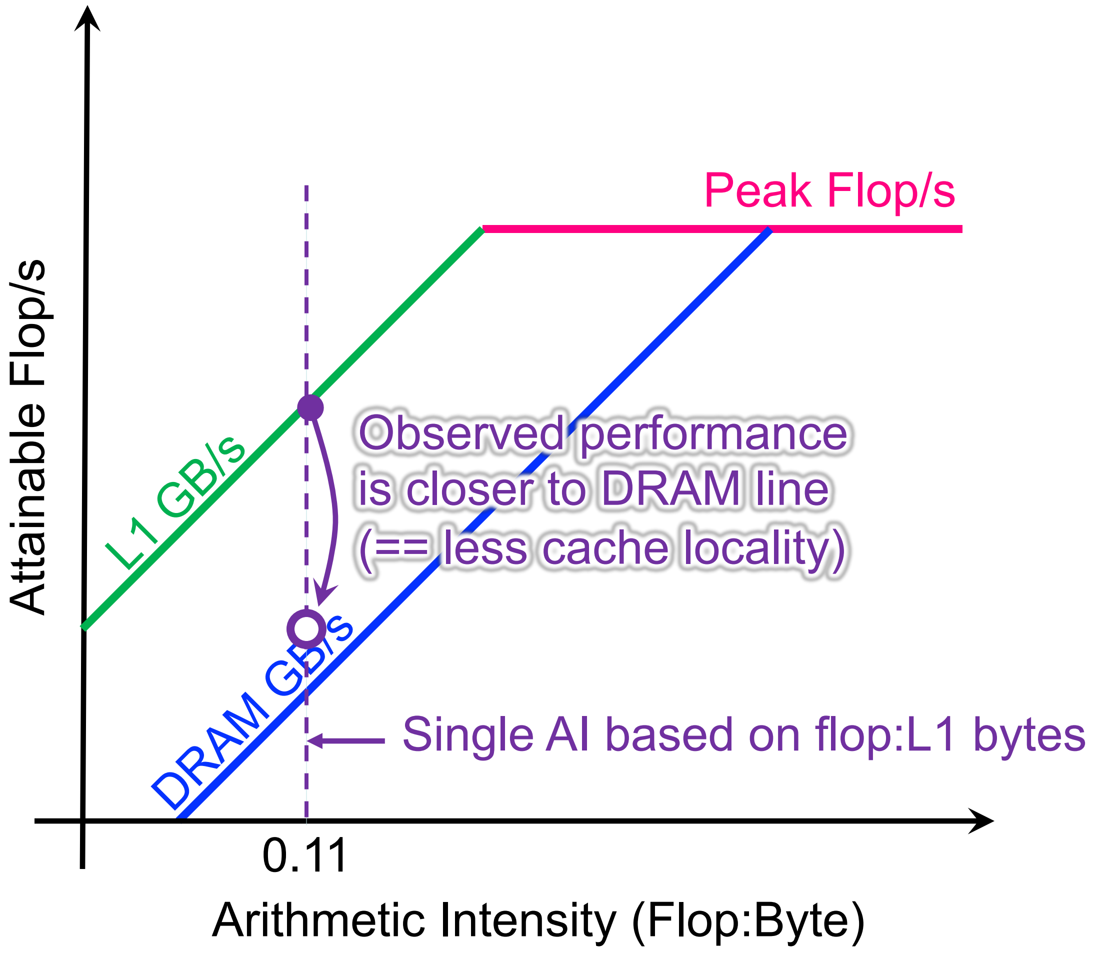
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Actual **observed** performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline



Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes

Questions?

Don’t forget to take the Survey…

http://bit.ly/sc18-eval

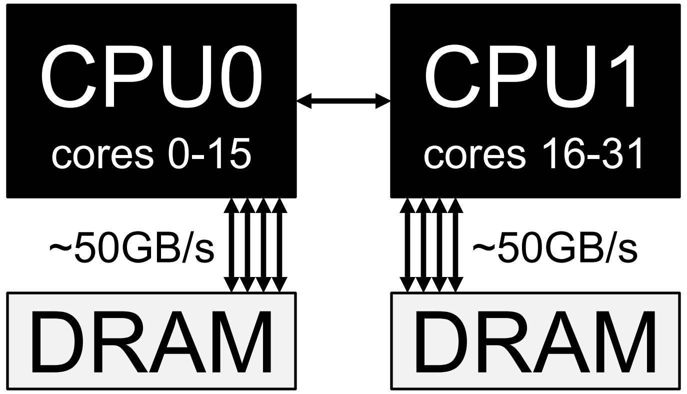
Backup

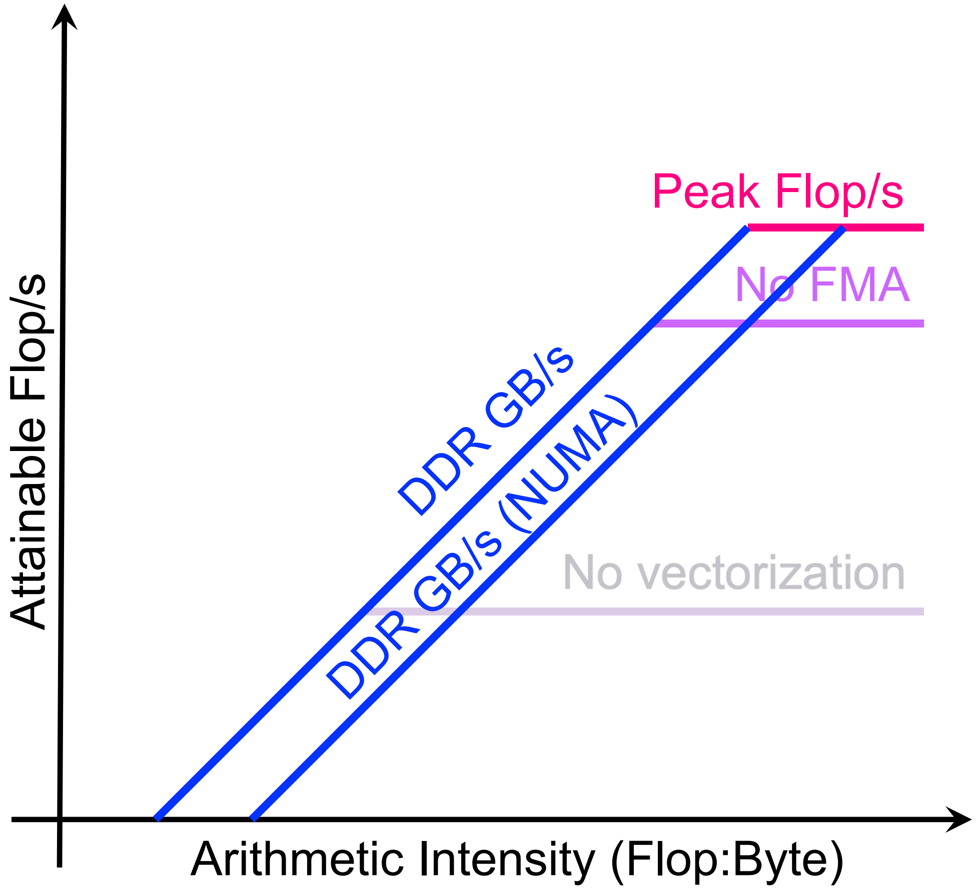
Refining Roofline:

NUMA

NUMA Effects

- Cori’s Haswell nodes are built from 2 Xeon processors (sockets)
  - Memory attached to each socket (fast)
  - Interconnect that allows remote memory access (slow == NUMA)
  - Improper memory allocation can result in more than a 2x performance penalty



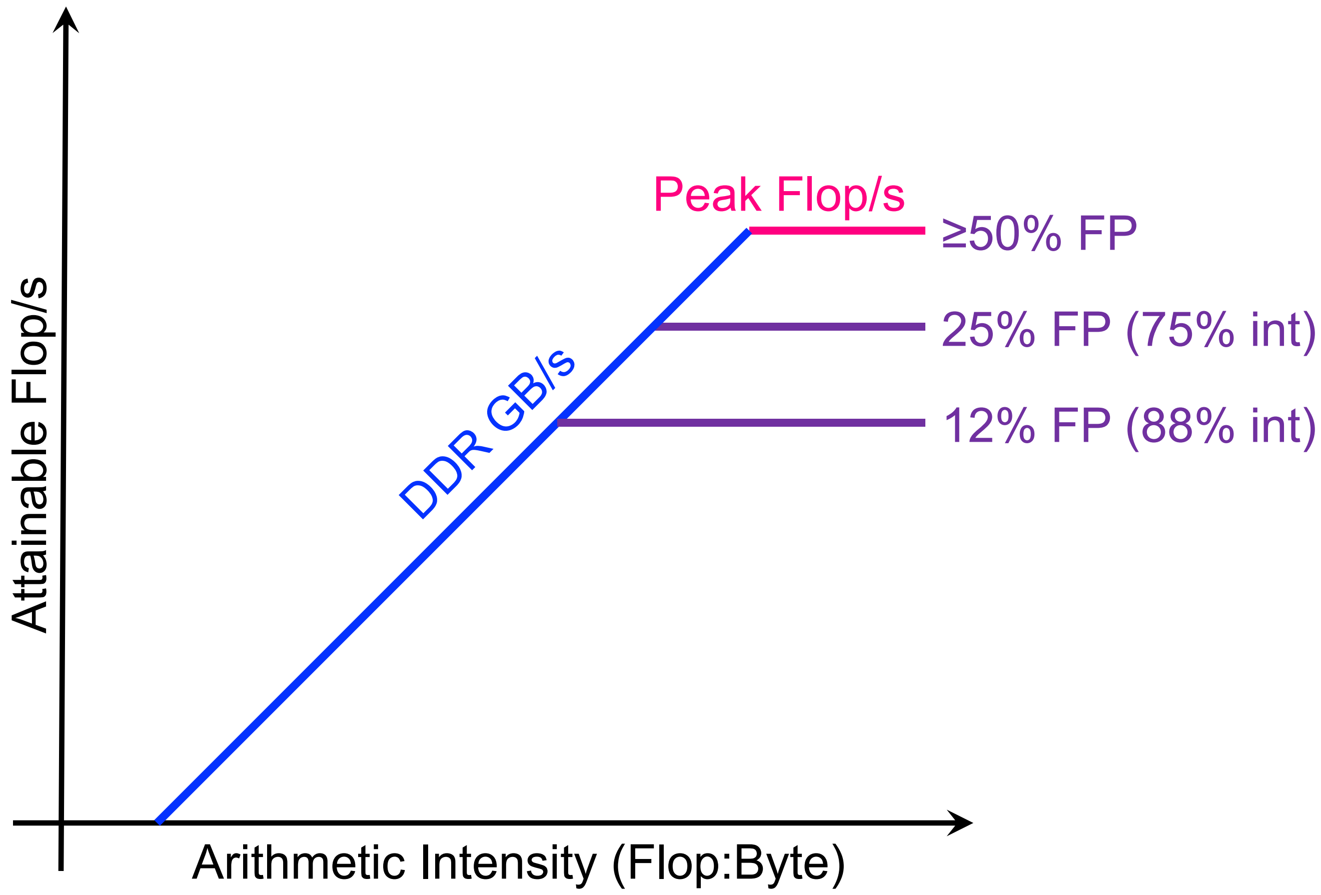


Refining Roofline:

Instruction Issue Bandwidth

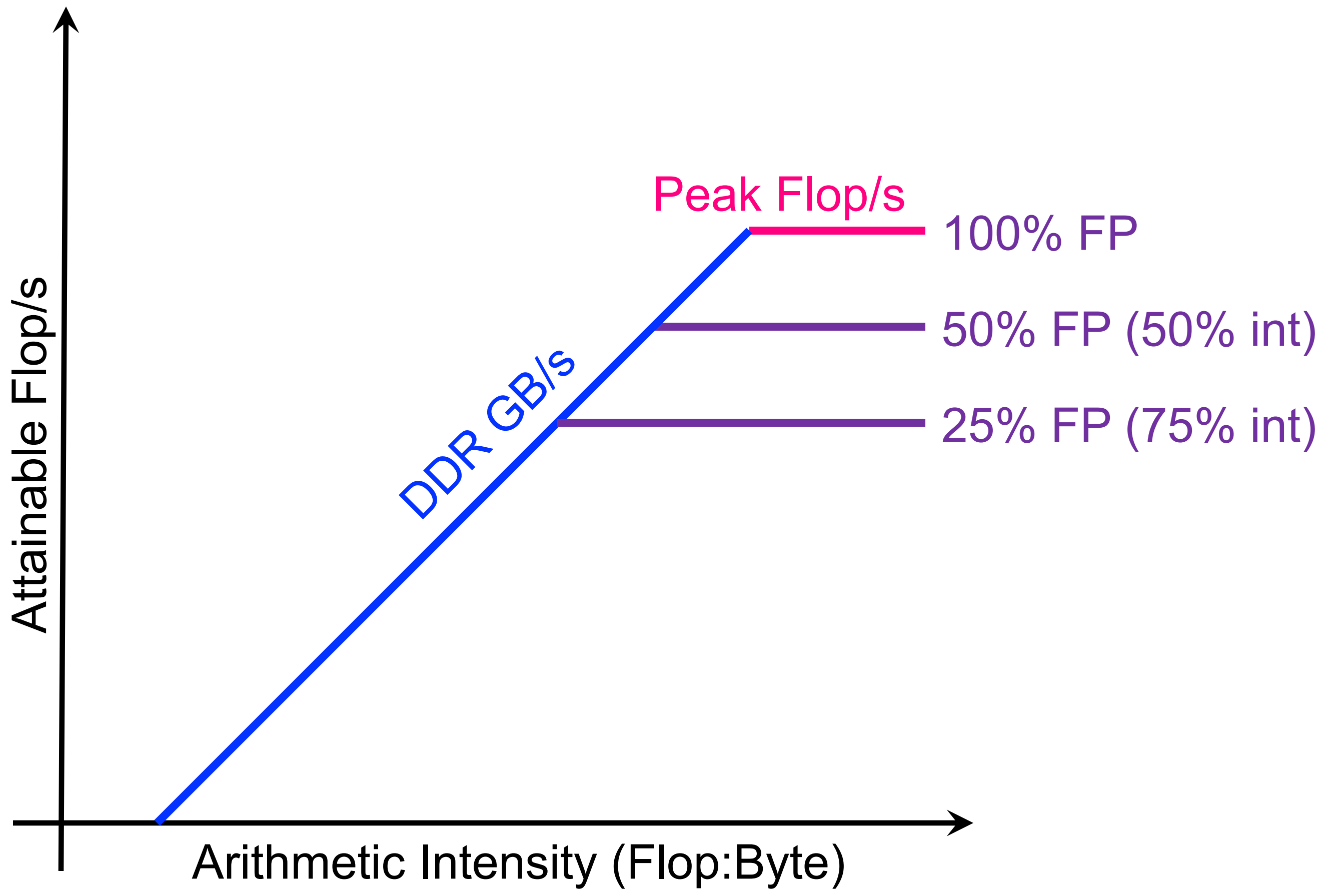
Superscalar vs. Instruction mix

- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance



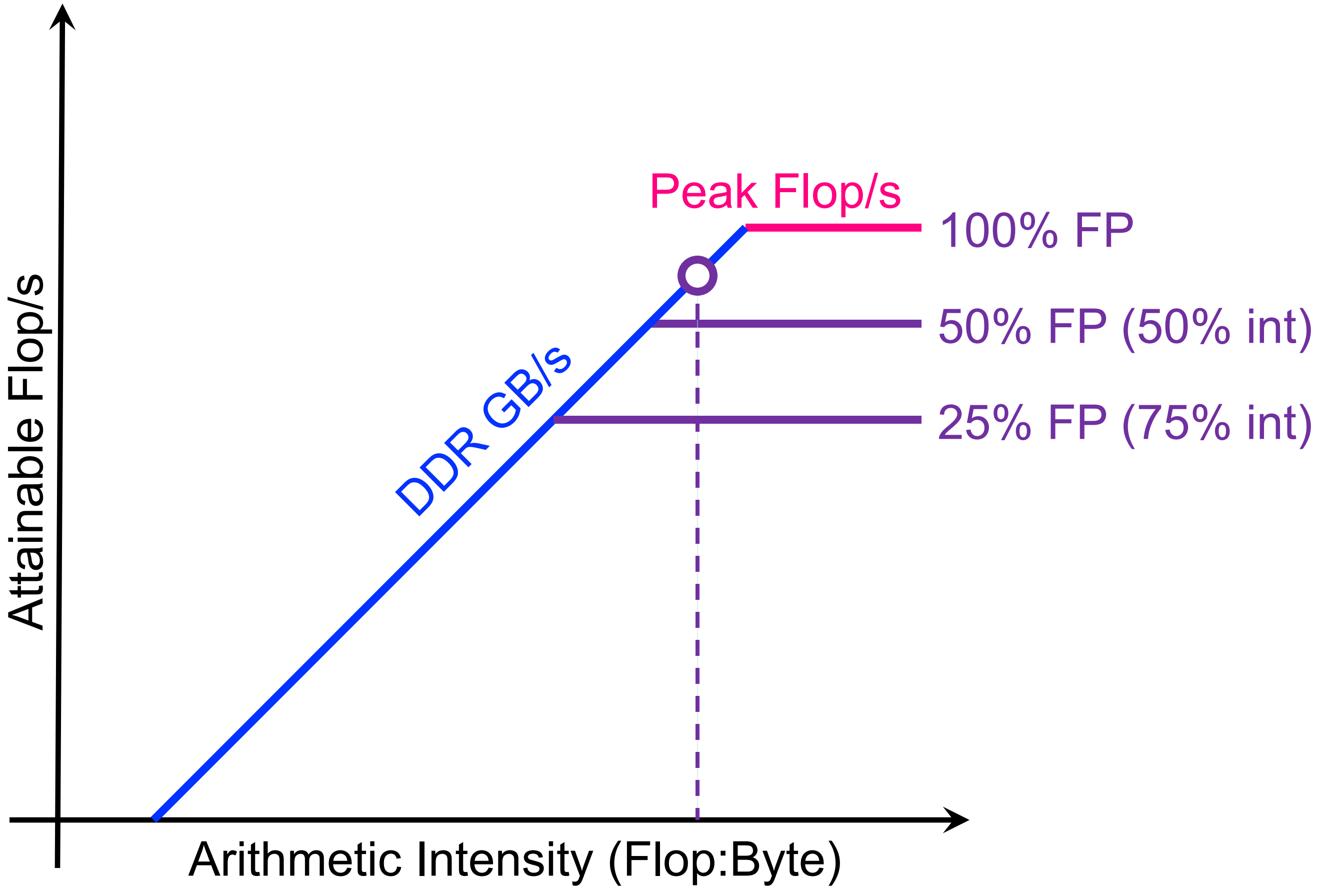
Superscalar vs. Instruction mix

- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- e.g. KNL
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



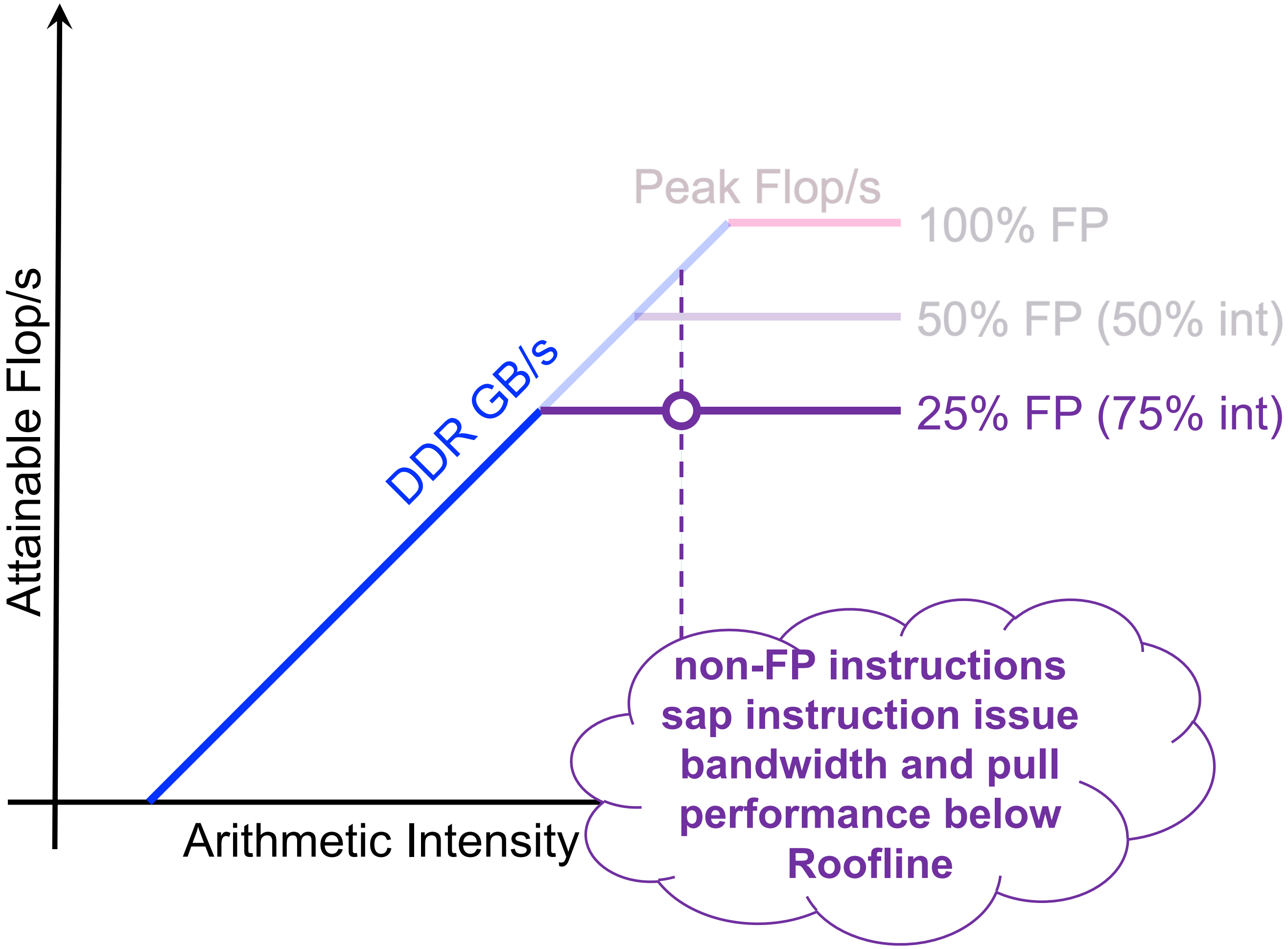
Superscalar vs. instruction mix

- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- e.g. KNL
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



Superscalar vs. instruction mix

- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- e.g. KNL
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



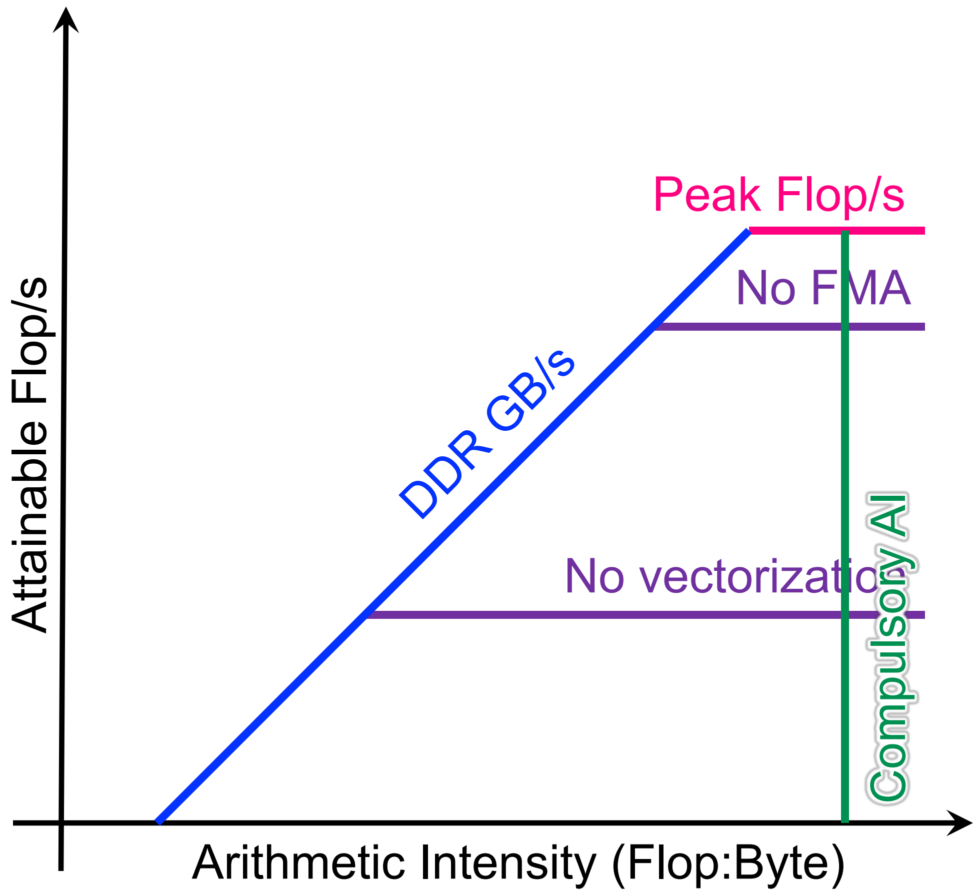
non-FP instructions sap instruction issue bandwidth and pull performance below Roofline

Refining Roofline:

Compulsory, Capacity, and Conflict misses

Locality Walls

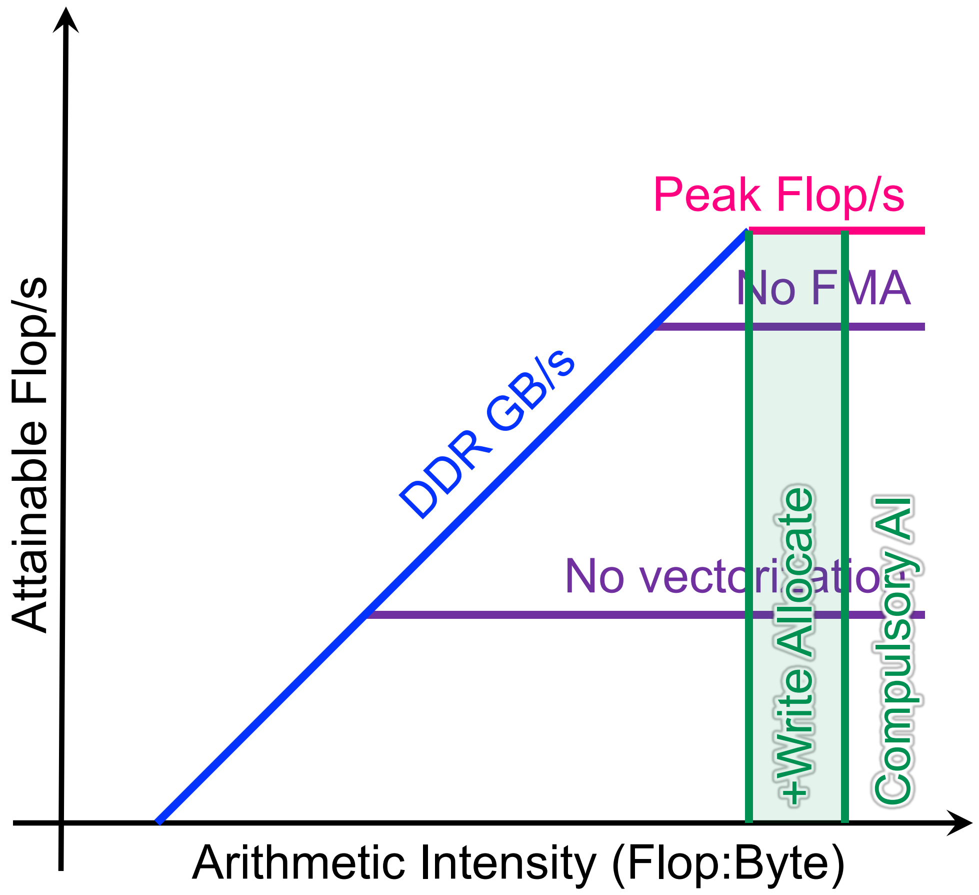
- Naively, we can bound AI using only compulsory cache misses



$$AI = \frac{\#Flop's}{Compulsory\ Misses}$$

Locality Walls

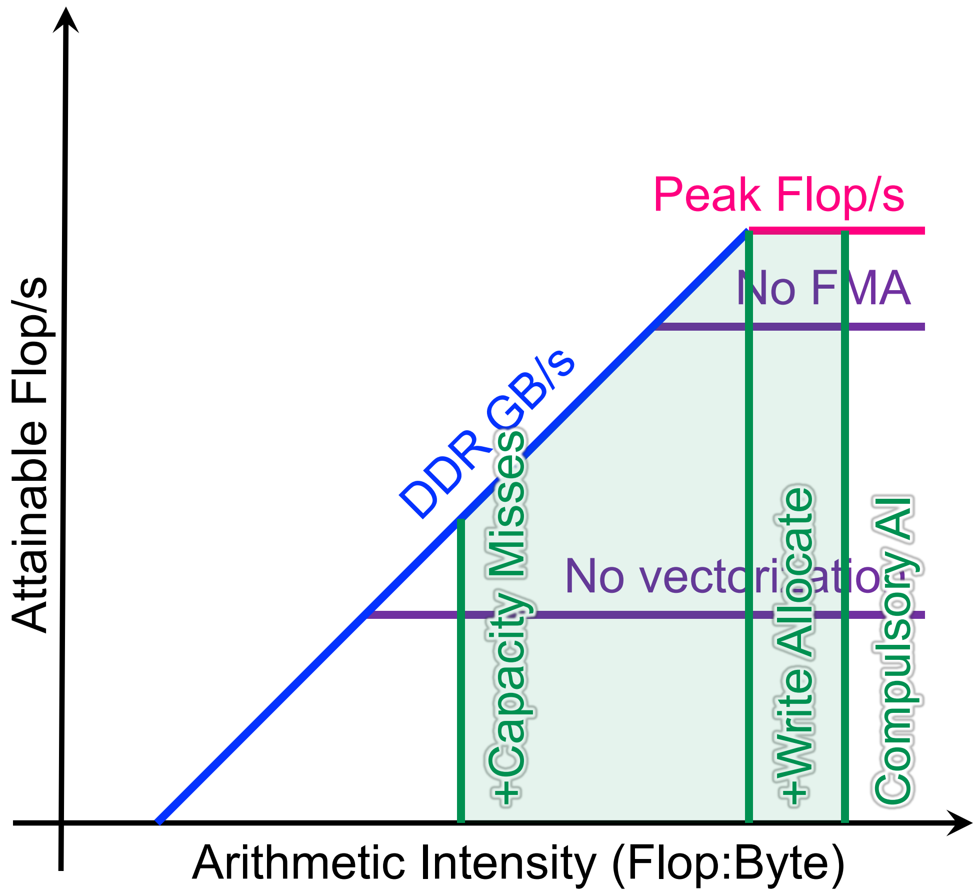
- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI



$$AI = \frac{\#Flop's}{\text{Compulsory Misses + Write Allocates}}$$

Locality Walls

- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI
- Cache capacity misses can have a huge penalty



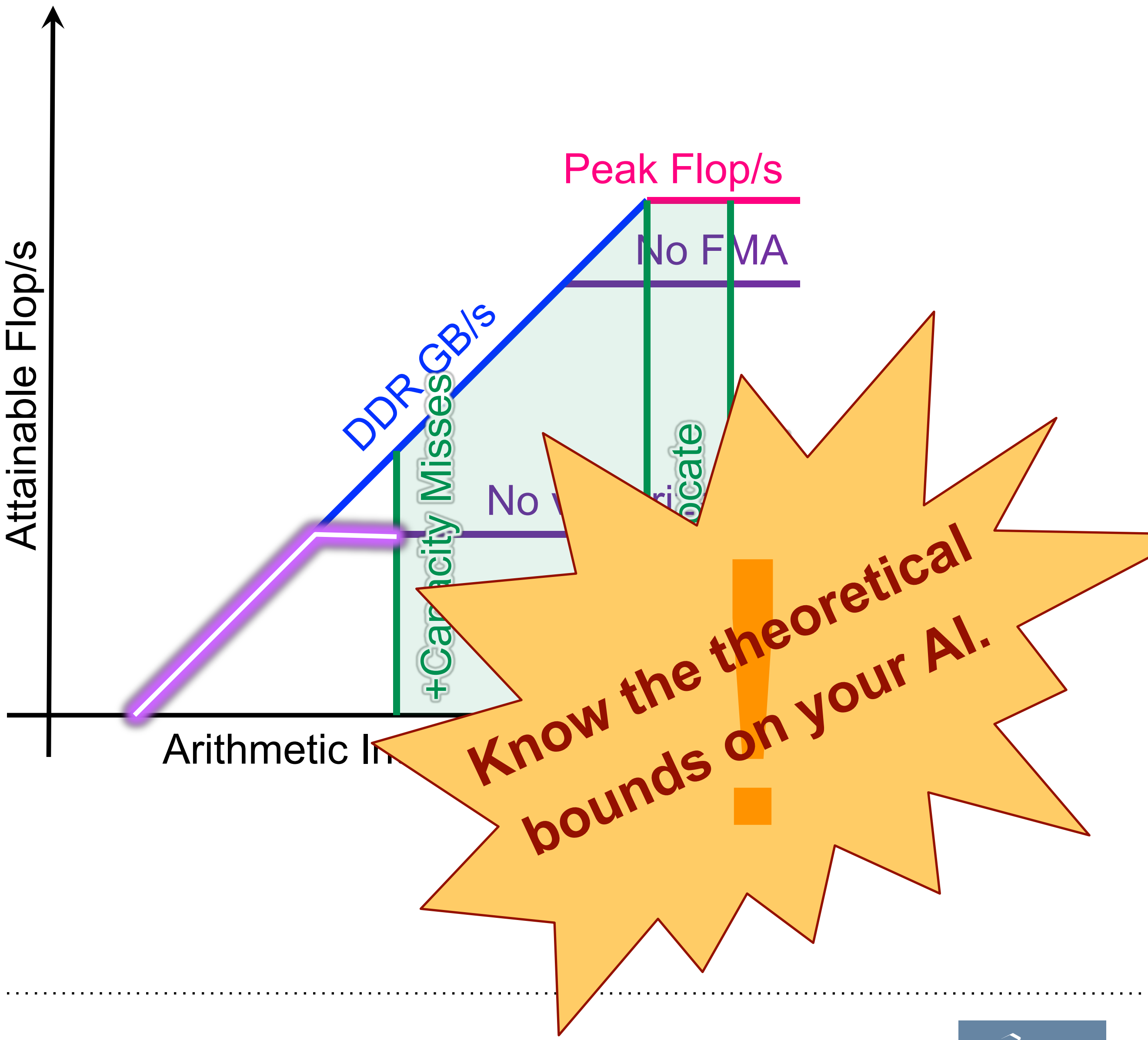
$$\text{AI} = \frac{\text{\#Flop's}}{\text{Compulsory Misses + Write Allocates + Capacity Misses}}$$

Locality Walls

- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI
- Cache capacity misses can have a huge penalty

➢ **Compute bound became memory bound**

$$AI = \frac{\#Flop's}{\text{Compulsory Misses + Write Allocates + Capacity Misses}}$$



Know the theoretical bounds on your AI.

LIKWID:

Performance Counters

LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters...
  - ✓ **Works on NERSC production systems**
  - ✓ Distills counters into user-friendly metrics (e.g. MCDRAM Bandwidth)
  - ✓ Minimal overhead (<1%)
  - ✓ Scalable in distributed memory (MPI-friendly)
  - ✓ Fast, high-level characterization
  - ✗ No timing breakdowns
  - ✗ Suffers from Garbage-in/Garbage Out

    (i.e. hardware counter must be sufficient and correct)

https://github.com/RRZE-HPC/likwid

http://www.nersc.gov/users/software/performance-and-debugging-tools/likwid