Performance Models
CS267 Lecture 4

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spr2020/

Acknowledgements

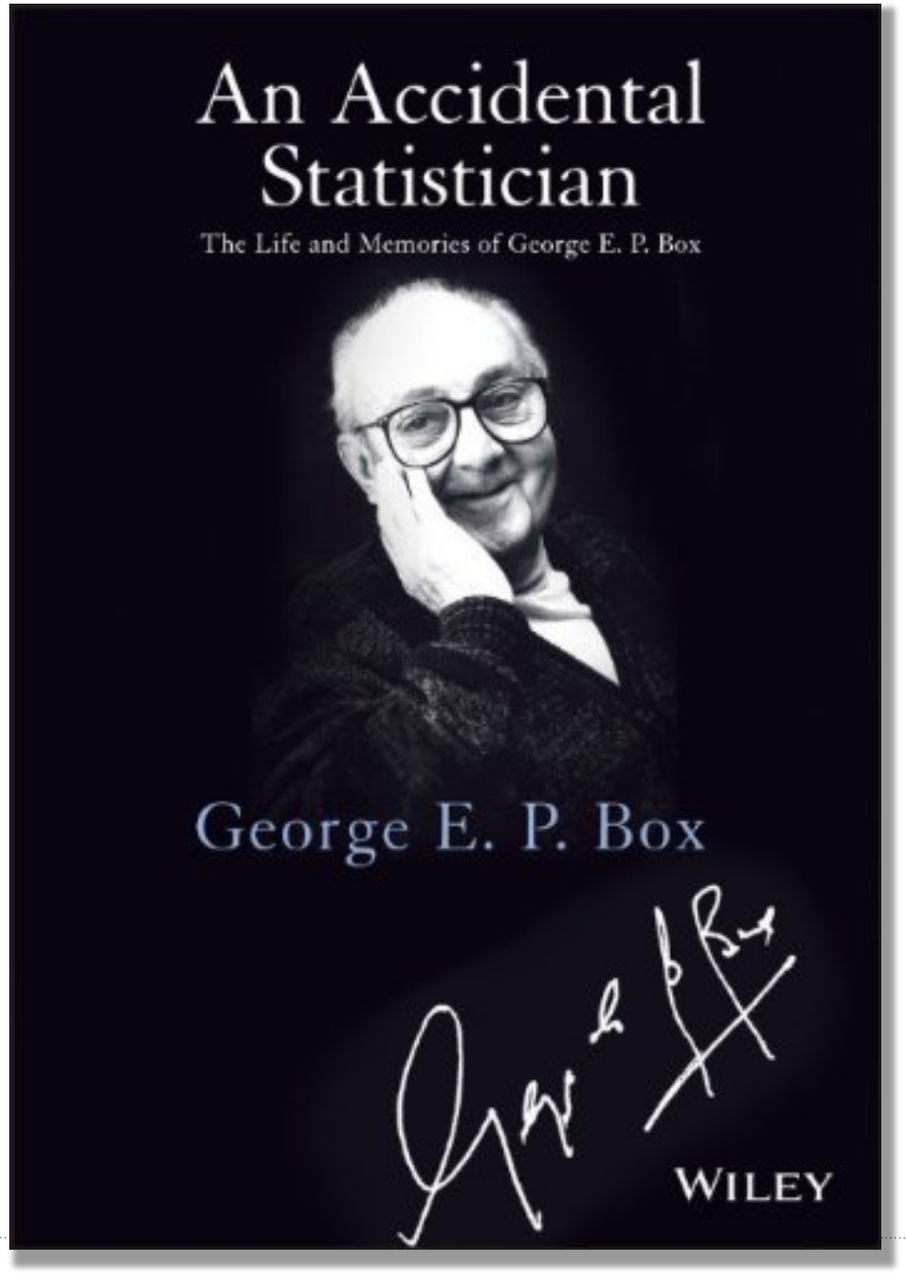
Many of the Rooflines slides are from Sam Williams, who invented the model.

He also acknowledges the following for these slides: Jack Deslippe, Charlene Yang, Doug Doerfler, Matt Cordery, Khaled Ibrahim, Lenny Oliker, Protonu Basu (now Facebook), Terry Ligocki, Brian Van Straalen (LBNL), Linda Lo (formerly Utah), Zakhar Matveev (Intel), Roman Belenov (Intel)

Some image are from a paper by Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, Markus Puschel (ETH)

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.



all models are wrong,
but some are useful

- George Box

What makes a performance model useful?

Simple enough to give insight

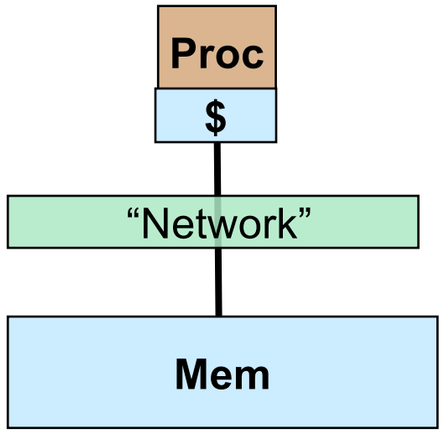
Accurate enough to be meaningful

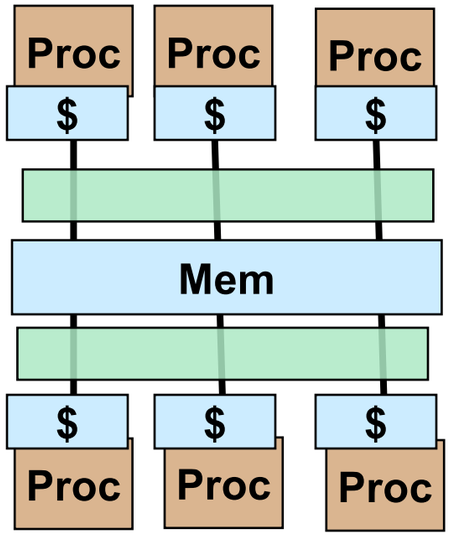
Why Use Performance Models or Tools?

- Understand performance behavior
  - Differences between Architectures, Programming Models, implementations, etc.
- 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.
- Identify performance bottlenecks
- Motivate better software optimizations
- **Determine when we're done optimizing**
  - Assess performance relative to machine capabilities
  - Motivate need for algorithmic changes

Serial and Shared Memory Machines

Two types of machines so far in this class

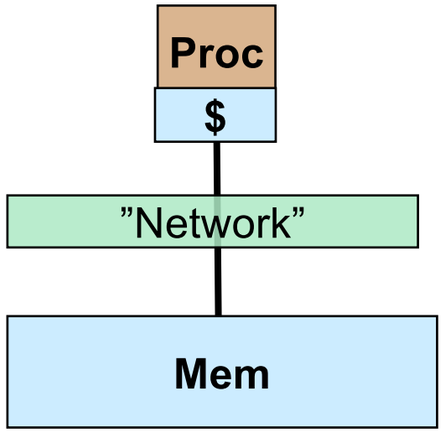


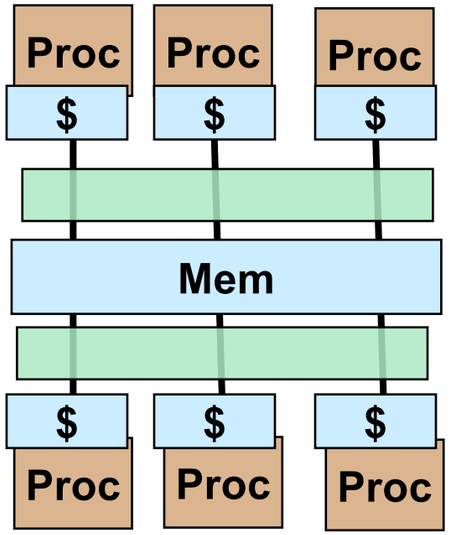


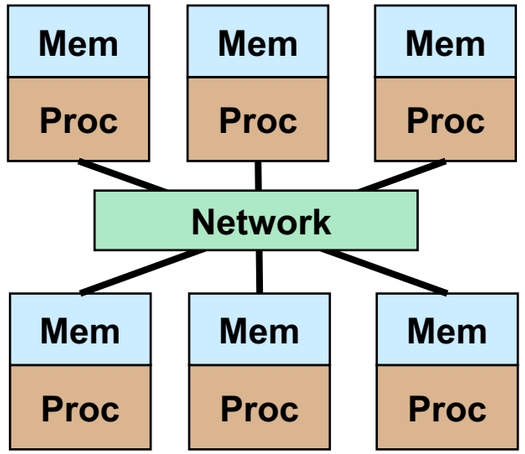
Sequential Machine	Shared Memory
Single processor	Processors execute own instruction stream
Reads and writes memory	Communicate by reading/writing memory

- Critical performance issues
- Slow and fast memory (e.g., cache = $)
  - Latency – cost for first Byte
  - Bandwidth – rate Bytes/sec
- Parallelism
  - Instruction level parallelism (ILP)
  - SIMD instructions
  - Multiple processor cores

Parallel Machines and Programming





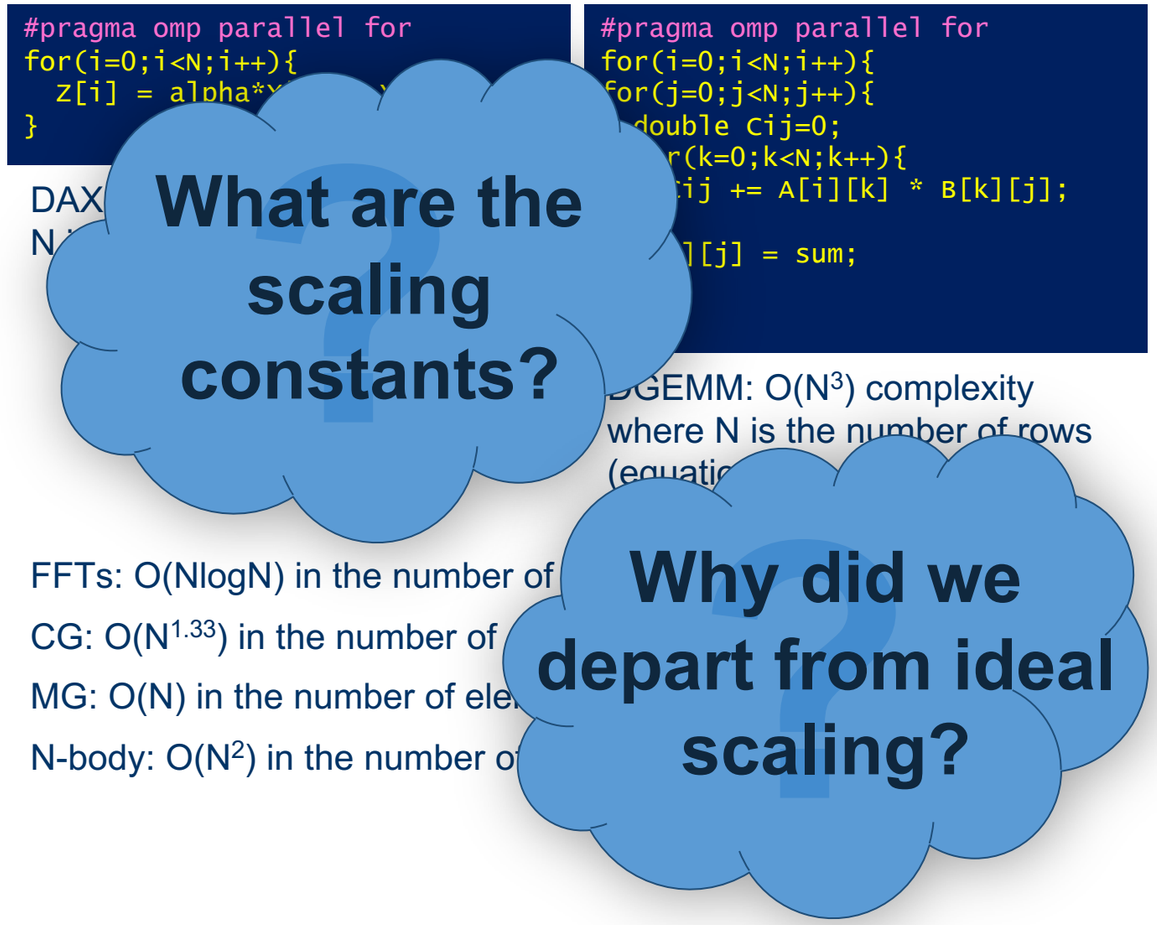


Shared Memory	Shared Memory	Distributed Memory
Processors execute own instruction stream	Processors execute own instruction stream	Processors execute own instruction stream
Communicate by reading/writing memory	Communicate by reading/writing memory	Communicate by sending messages (MPI Send/Receive)

Focus still here today

Model #1: Computational Complexity

- Assume run time is correlated with the number of operations (e.g. FP ops)
- Users define parameterize their algorithms, solvers, kernels
- Count the number of operations as a function of those parameters
- Demonstrate run time is correlated with those parameters



You will see the other algorithms next we in overview and throughout the semester in detail.

Computational Complexity Model

- Machine performance = peak flop rate
  - Intel Haswell (Cori): 36.8 Gflop/s/core
    1.2 Tflop/s/node
  - Intel KNL (Cori): 44.8 Gflop/s/core
    3 Tflop/s/node

Let γ be 1 / peak_flop/s
γ = 27 ns for a core
.9 ns for a node (32 cores)

γ = 23 ns
.3 nanosection (68 cores)

- Application cost = # flops
  - Matrix multiplication = $2*n^3$ Flops
  - Matrix vector mult = $2*n^2$ Flops

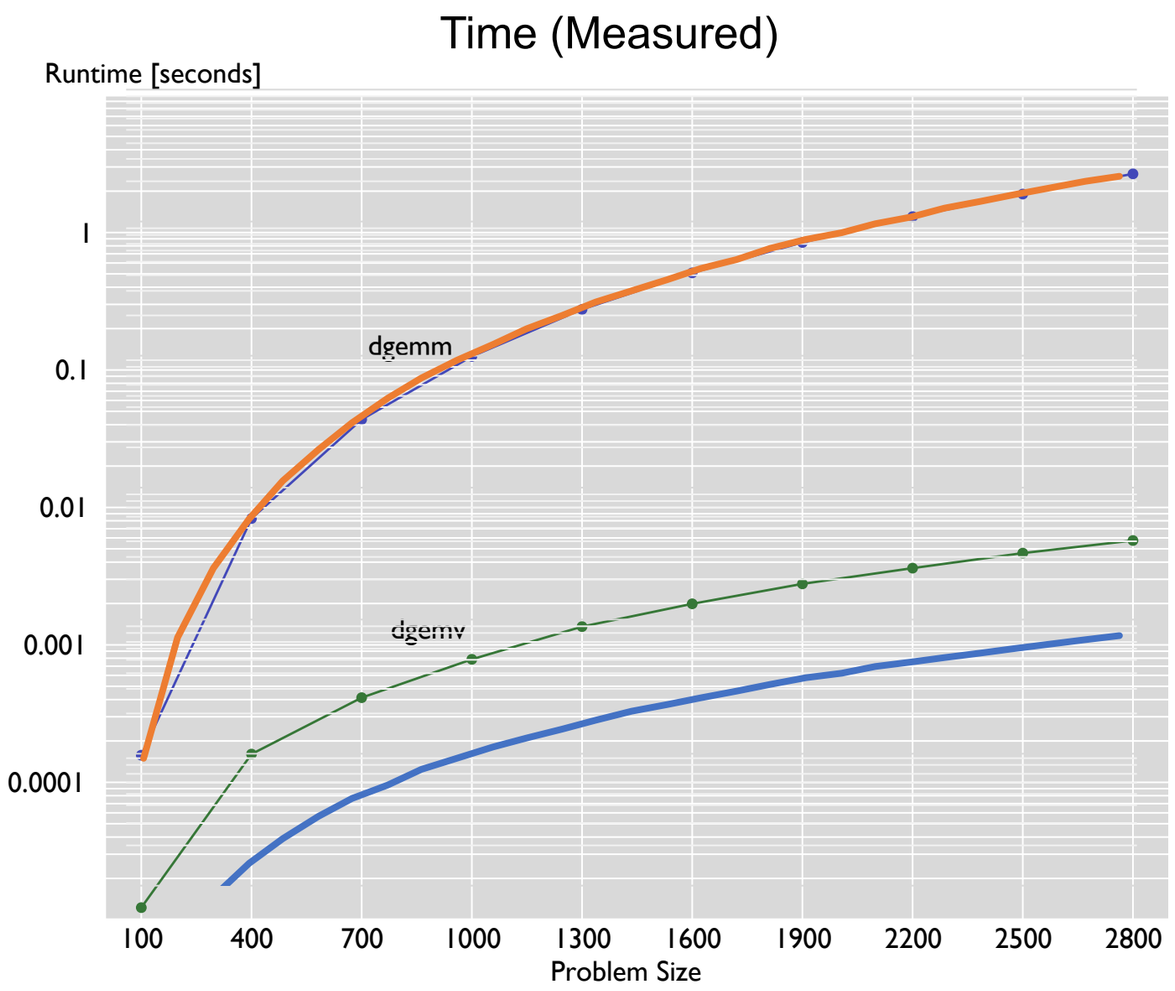
Time = $\gamma * $ # flops

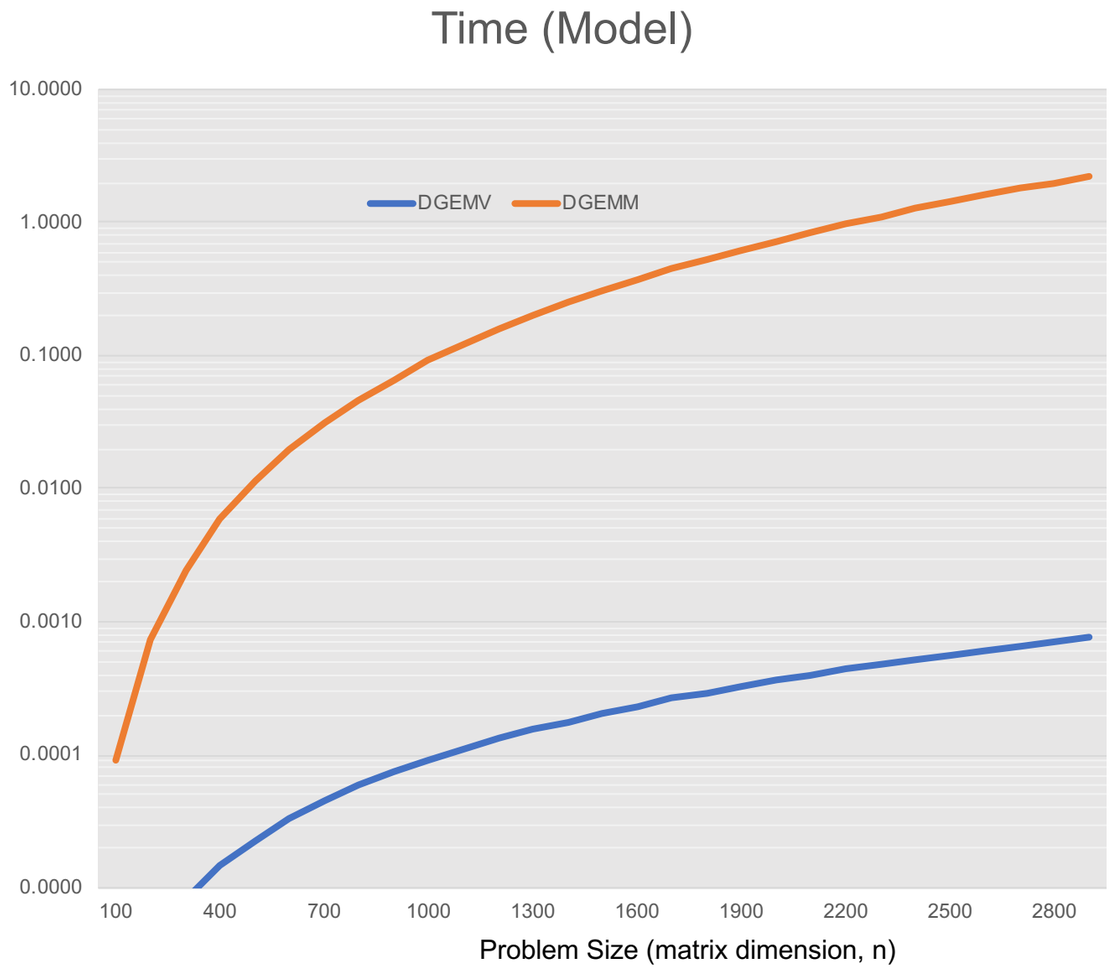
- We need the constants, not just $O(n^3)$ to predict actual running time

(Beware "flops" can be
ops *or* op rate
depending on context)

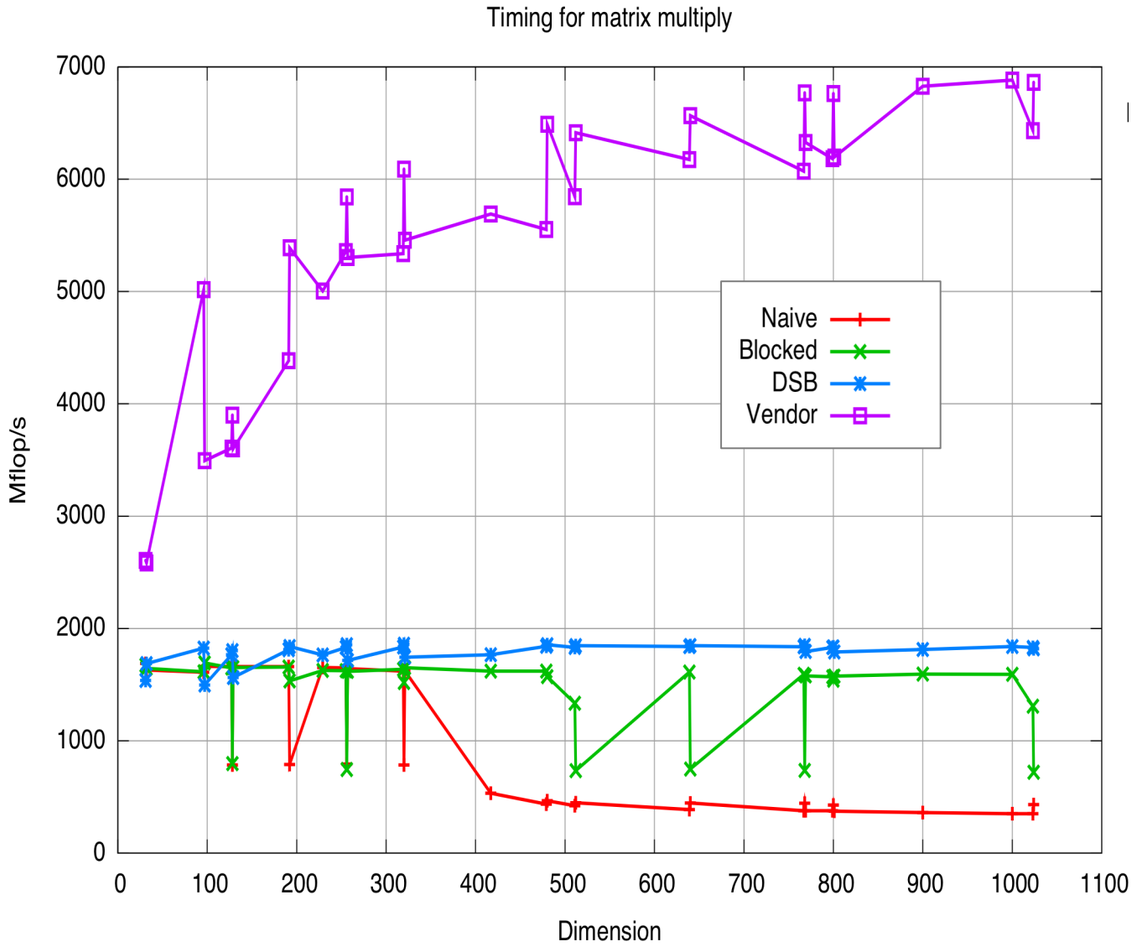
How good is this as a model?

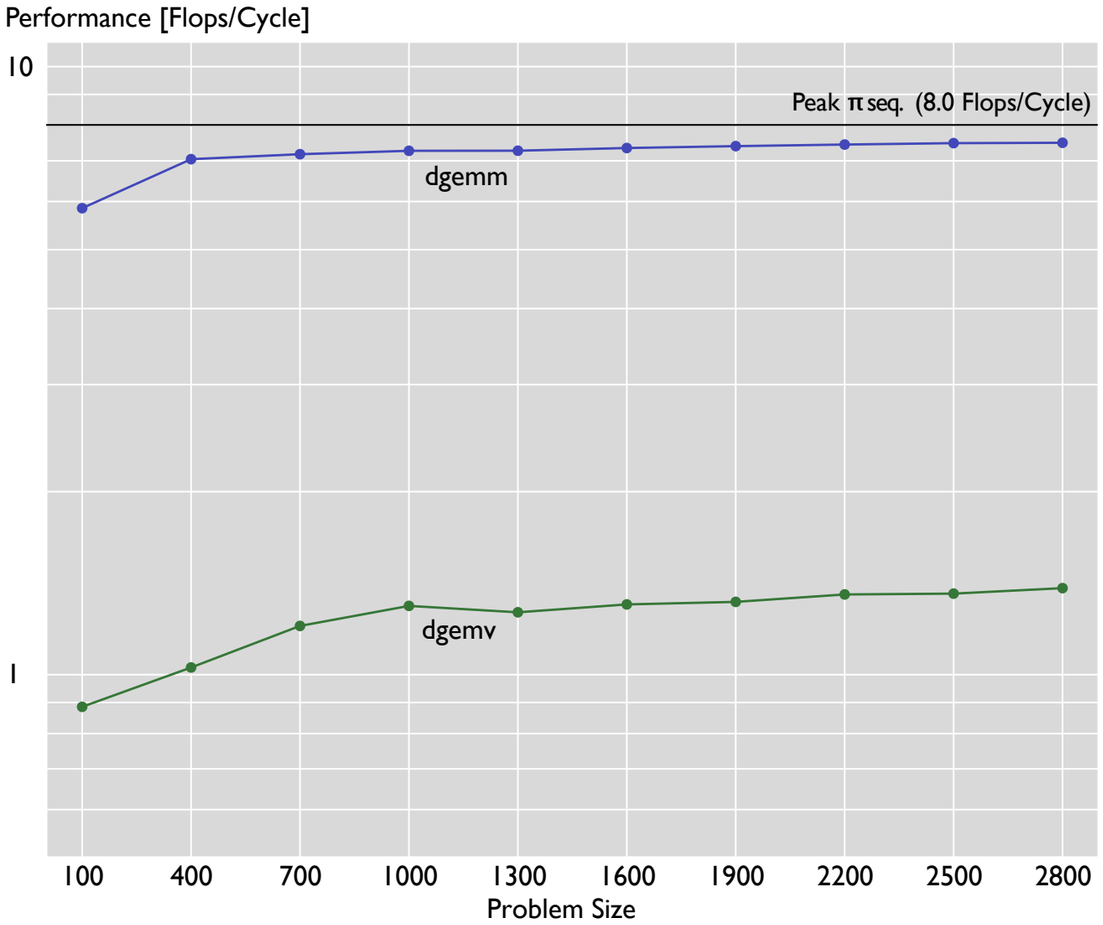
Peak flops / sec as a machine model, with computational complexity as an application model





How good is flop/s as a model?



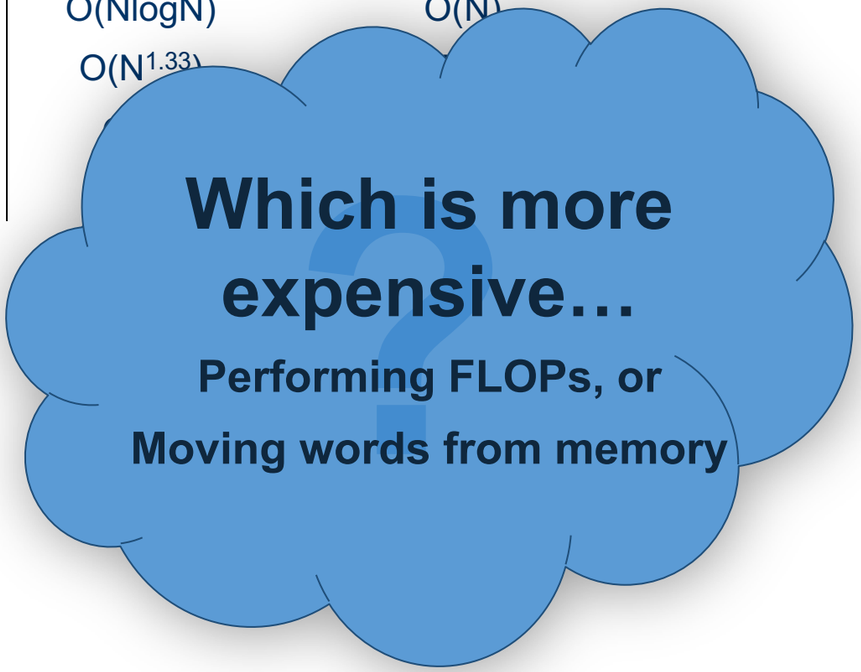


What's a better model for DGEMV (Matrix-vector multiply)?

Data Movement Complexity

- Assume run time is correlated with the amount of data accessed (or moved)
- Easy to calculate amount of data accessed… count array accesses
- Data moved is more complex as it requires understanding cache behavior…
  - Compulsory[1] data movement (array sizes) is a good initial guess…
  - … but needs refinement for the effects of finite cache capacities

Operation	FLOPs	Data
DAXPY	O(N)	O(N)
DGEMV	$O(N^2)$	$O(N^2)$
DGEMM	$O(N^3)$	$O(N^2)$
FFTs	O(NlogN)	O(N)
CG	$O(N^{1.33})$	
MG		
N-body		



[1]Hill et al, "Evaluating Associativity in CPU Caches", IEEE Trans. Comput., 1989.

Machine Balance and Arithmetic Intensity

- Data movement and computation can operate at different rates
- We define machine balance as the ratio of…

$$\textbf{Balance} = \frac{\textbf{Peak DP FLOP/s}}{\textbf{Peak Bandwidth}}$$

- …and computational / arithmetic intensity (CI/AI) as the ratio of…

$$\textbf{AI} = \frac{\textbf{FLOPs Performed}}{\textbf{Data Moved}}$$

Operation	FLOPs	Data	AI (ideal)
DAXPY	O(N)	O(N)	O(1)
DGEMV	$O(N^2)$	$O(N^2$	O(1)
DGEMM	(^{3})		O(N)
FFTs	O(N		O(logN)
CG	O(N		
MG			O(1)
N-body			O(N)

Kernels with AI less than machine balance are ultimately bandwidth limited

Computational Depth

- Multicore processors and even single core depend on parallelism
- Some algorithms are have some serial path that is inherent
- The **depth** is the longest dependence chain

➢ **Think of running on an infinite number of processor with no OMP, communication, or loop overheads**

Operation	FLOPs	Data	AI (Ideal)	Depth
DAXPY	O(N)	O(N)	1)	O(1)
DGEMV	O(N^2)	O	1)	O(logN)
DGEMM	O(		N)	O(logN)
FFTs	O(Nlo			O(logN)
CG	O(N1			logN)
MG				O(logN)
N-body				O(logN)

Overheads can dominate at high concurrency or small problems

Distributed Memory Performance Modeling

- In distributed memory, one communicates by sending messages between processors.
- Messaging time can be constrained by several components…
  - Overhead (CPU time to send/receive a message)
  - Latency (time message is in the network; can be hidden)
  - Message throughput (rate at which one can send small messages… messages/second)
  - Bandwidth (rate one can send large messages… GBytes/s)
- Bandwidths and latencies are further constrained by the interplay of network architecture and contention
- Distributed memory versions of our algorithms can be differently stressed by these components depending on N and P (#processors)

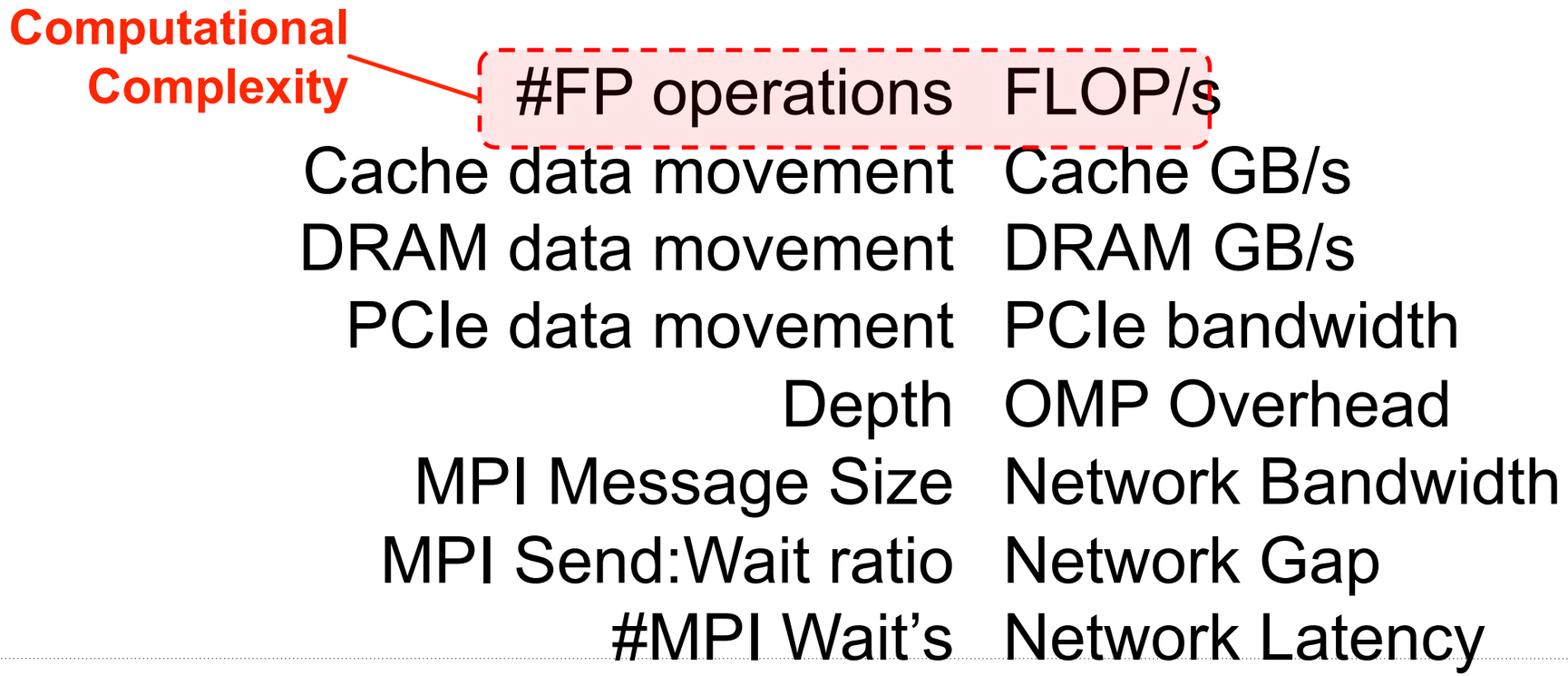
Performance Models

- Many different components can contribute to kernel run time.
- Some are characteristics of the application, some are characteristics of the machine, and some are both (memory access pattern + caches).

#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…



Performance Models

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

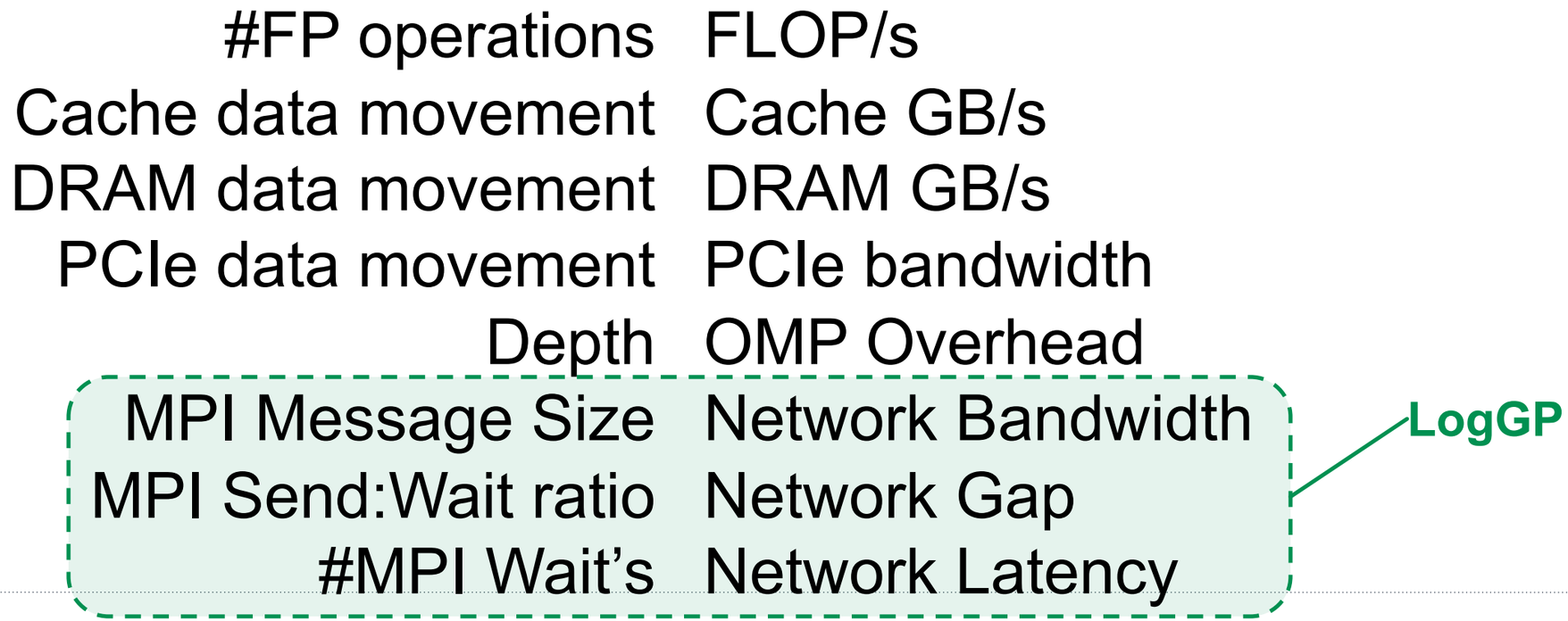
#FP operations	FLOP/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

LogP

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

Performance Models

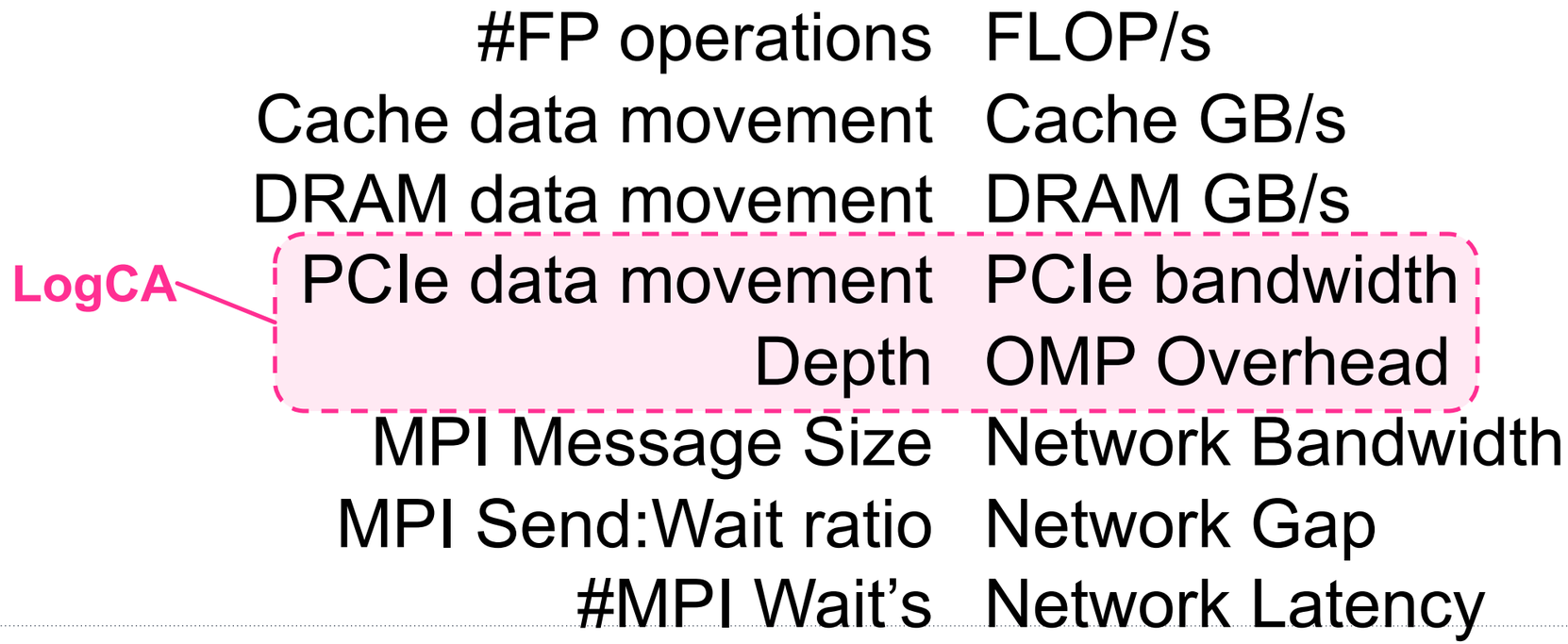
- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.



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.



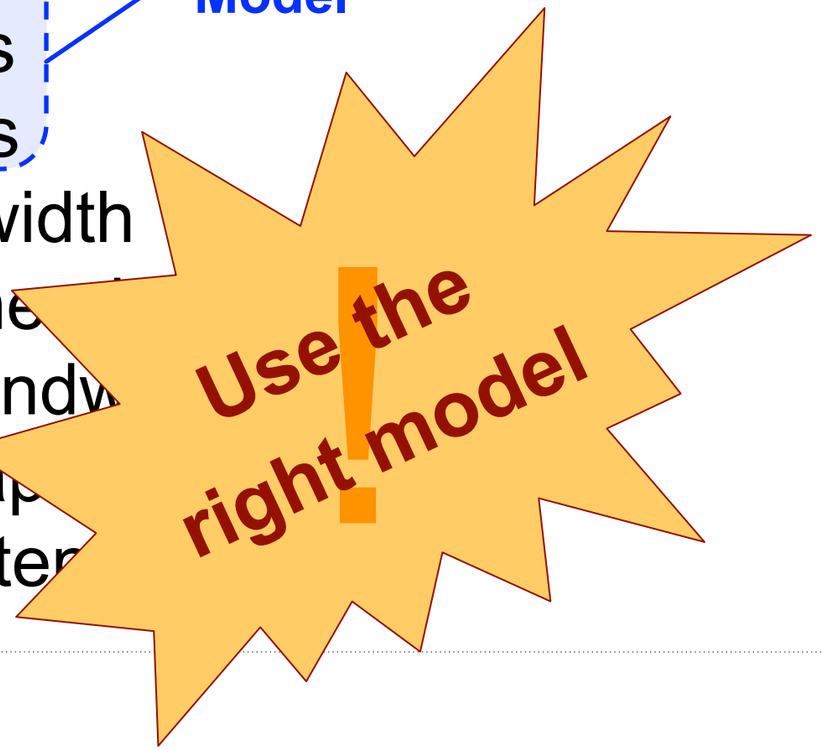
Bin Altaf et al, "LogCA: A High-Level Performance Model for Hardware Accelerators", ISCA, 2017.

Performance Models

- Performance models approximate the system by dominant terms.
- Which things matter most (missing # cores, mem latency…)

#FP operations	FLOP/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhe
MPI Message Size	Network Bandw
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Later

Roofline Model



Implications of Architectural Evolution…

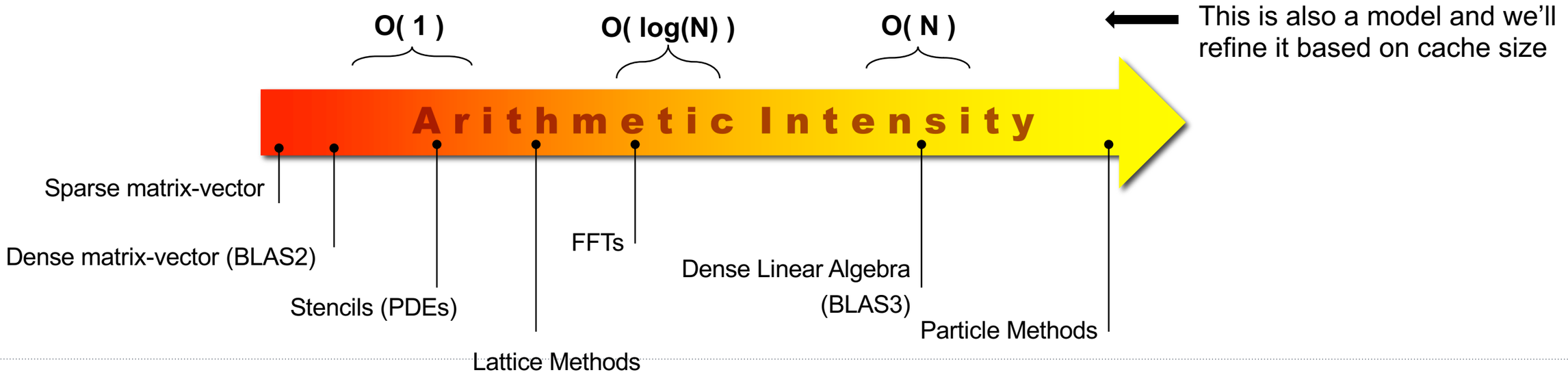
- Historically, many performance models and simulators tracked time to predict performance (i.e. counting seconds or counting cycles)

- 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**

Computational Intensity

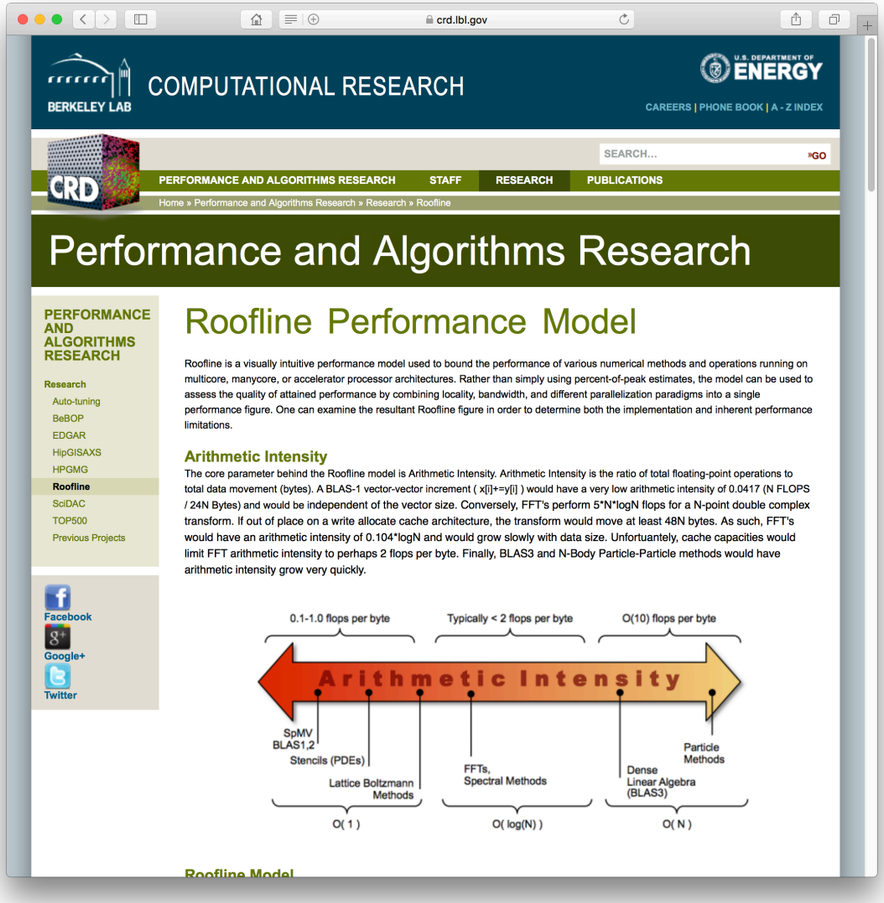
- **Computation Intensity (CI) ~ Total Flops / Total DRAM Bytes moved**
  Also called Arithmetic Intensive (AI) for numerical applications
- **Can we hit the bandwidth or flop limit?**

- **The rest is all about having and being able utilize enough concurrency**



Roofline Model

- **Roofline Model** is a throughput-oriented performance model
- Tracks rates not times
- Uses bound and bottleneck analysis
- Independent of architecture and hardware details (applies to CPUs, Multicore, GPUs, Google TPUs[1], etc…)



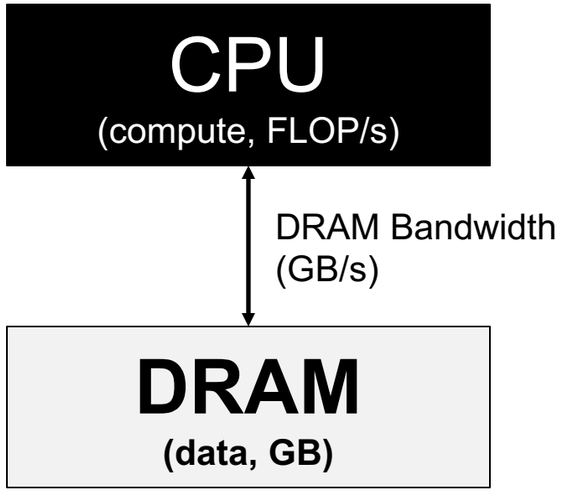
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

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

Introduction to the Roofline Model

(DRAM) Roofline

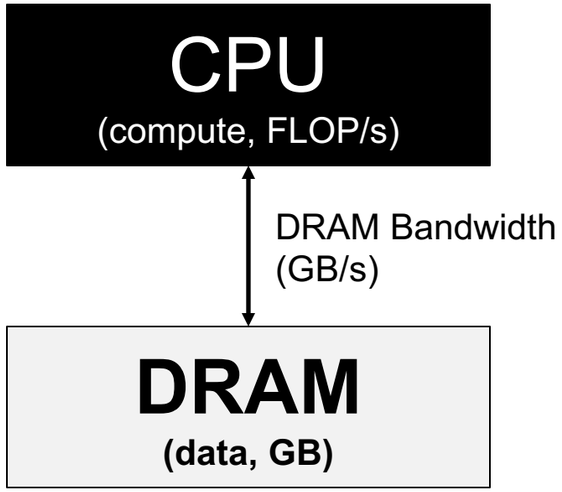
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

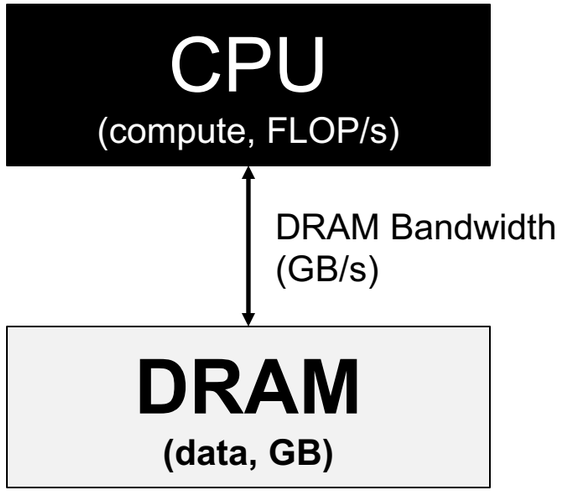
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

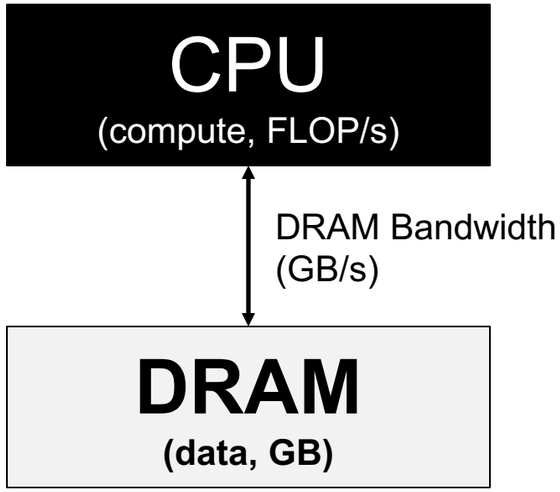
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

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



GFLOP/s = min { Peak GFLOP/s ; AI * Peak GB/s }

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

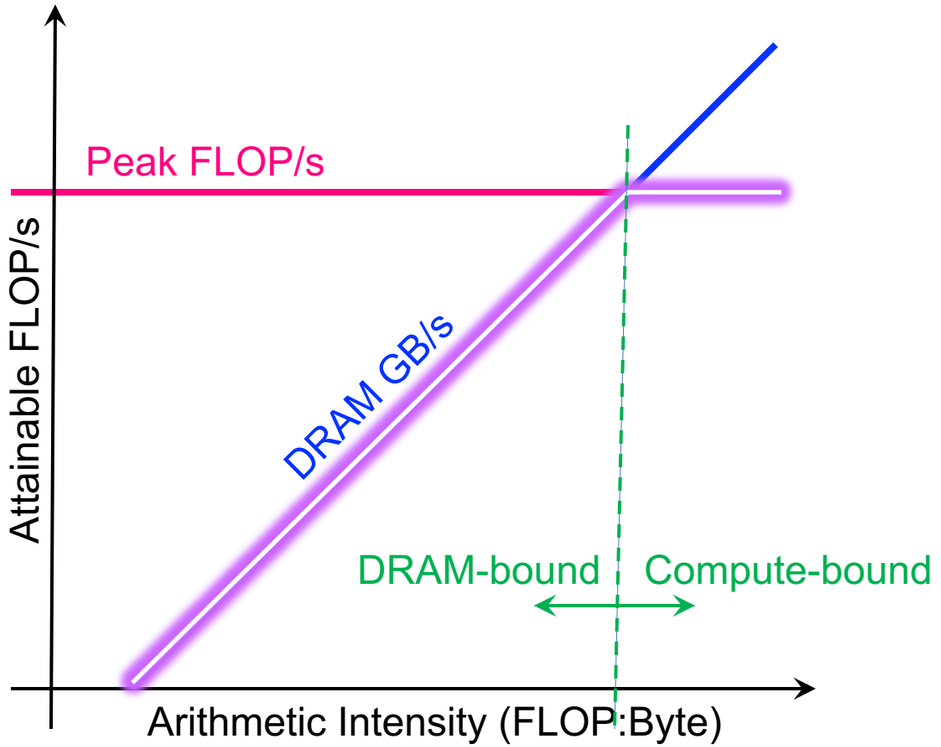
Arithmetic Intensity

- The most important concept in Roofline is **Arithmetic Intensity**

- 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)
  - Equal to ratio of sustained GFLOP/s to sustained GB/s (time cancels)

(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…)



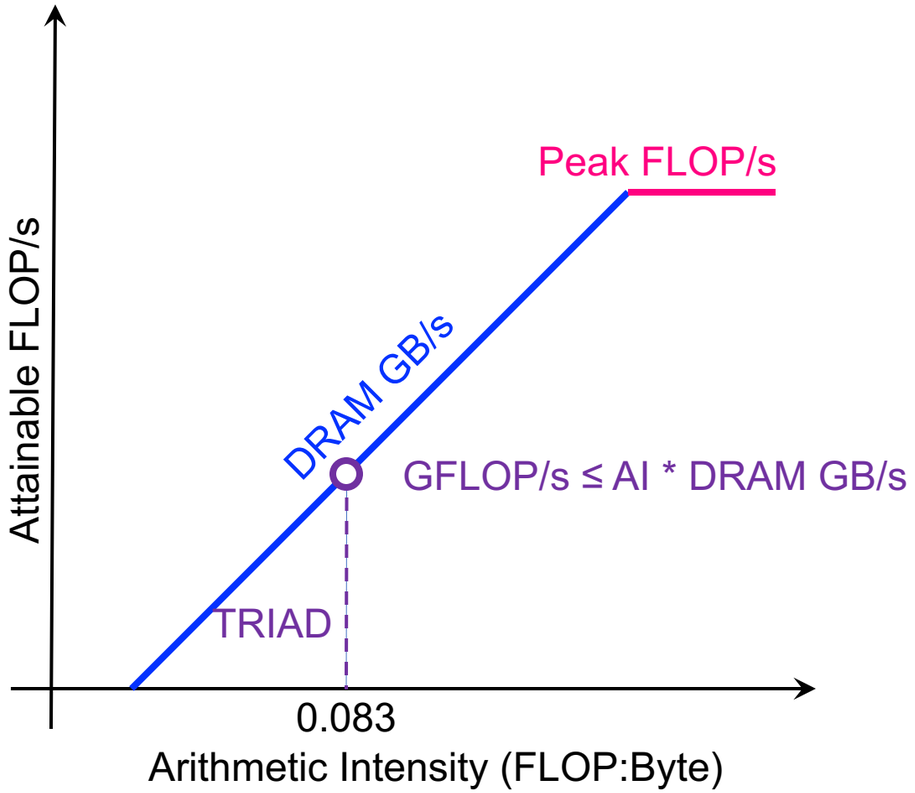
Roofline Example #1

- 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];
}
```

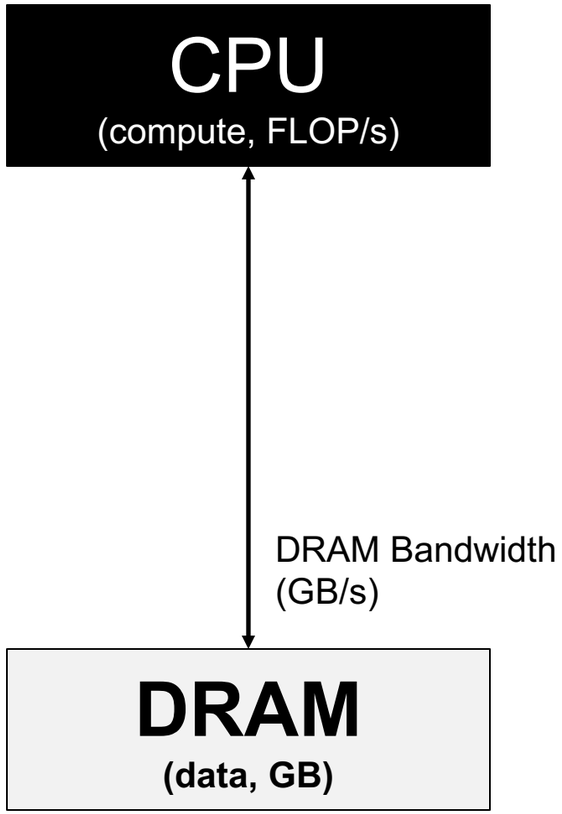
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 flops per byte == Memory bound**



Roofline Example #2

- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - **AI = 0.11 flops per byte (L1)**

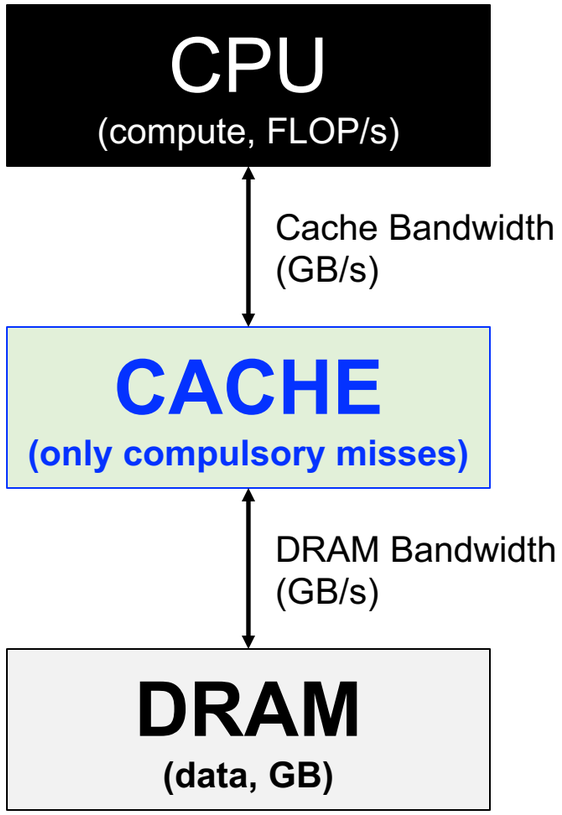
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                     + old[k  ][j  ][i-1]
                     + old[k  ][j  ][i+1]
                     + old[k  ][j-1][i  ]
                     + old[k  ][j+1][i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



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**

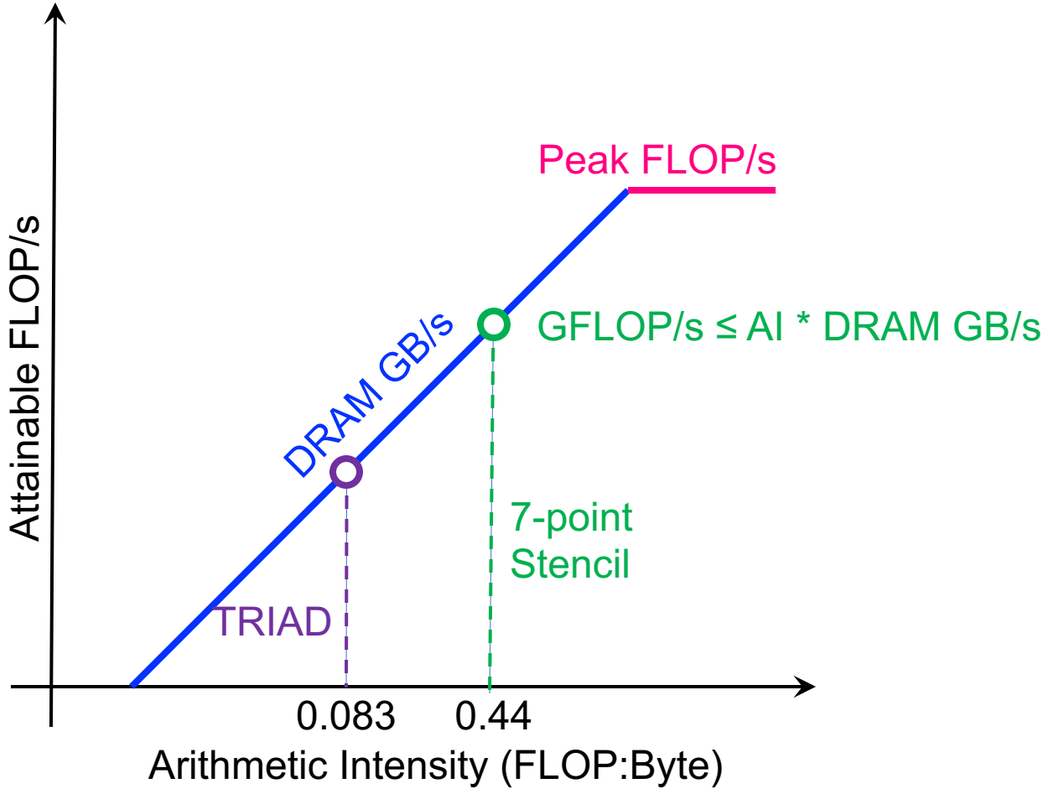
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
 new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                + old[k  ][j  ][i-1]
                + old[k  ][j  ][i+1]
                + old[k  ][j-1][i  ]
                + old[k  ][j+1][i  ]
                +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



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  ];
}}}
```



3 Minute Break

- By popular demand
- Will answer 1:1 questions and repeat any “interesting” ones to the group after the break

Question:

Will Performance Always Lie on the Roofline?

Can performance be below the Roofline?

- Analogous to stating that one can always attain either…
  - Peak Bandwidth
  - Peak FLOP/s

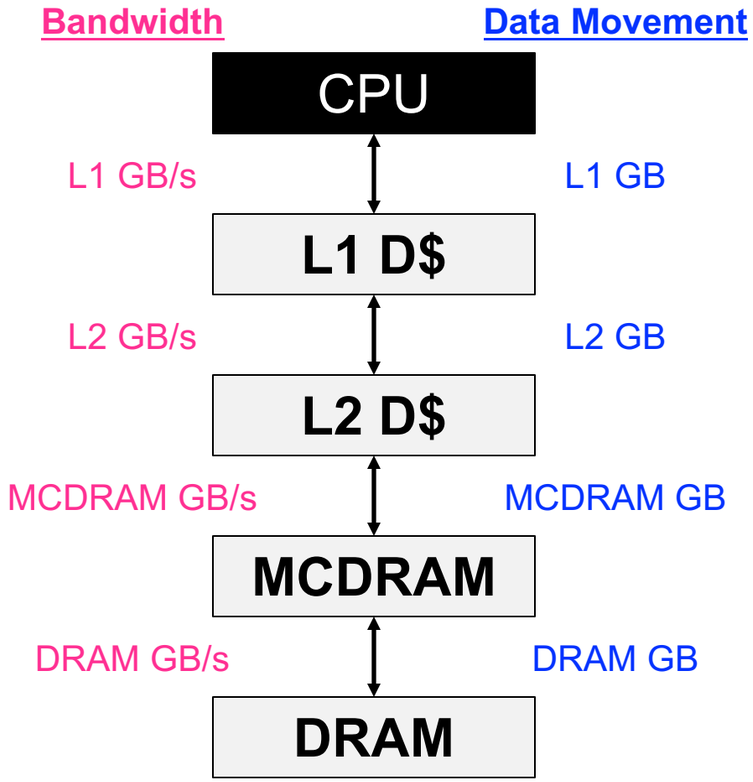
- **No, there can be other performance bottlenecks…**
  - Cache bandwidth / locality
  - Lack of vectorization / SIMDization
  - Load imbalance
  - …

Extending the Roofline:

Memory Hierarchy

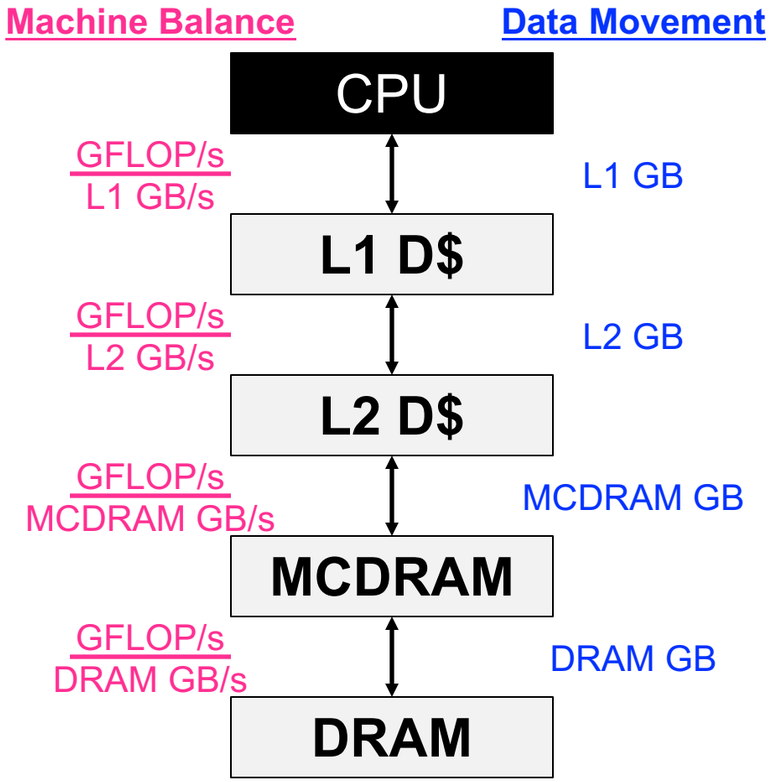
Hierarchical Roofline

- 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 unique peak and sustained bandwidths



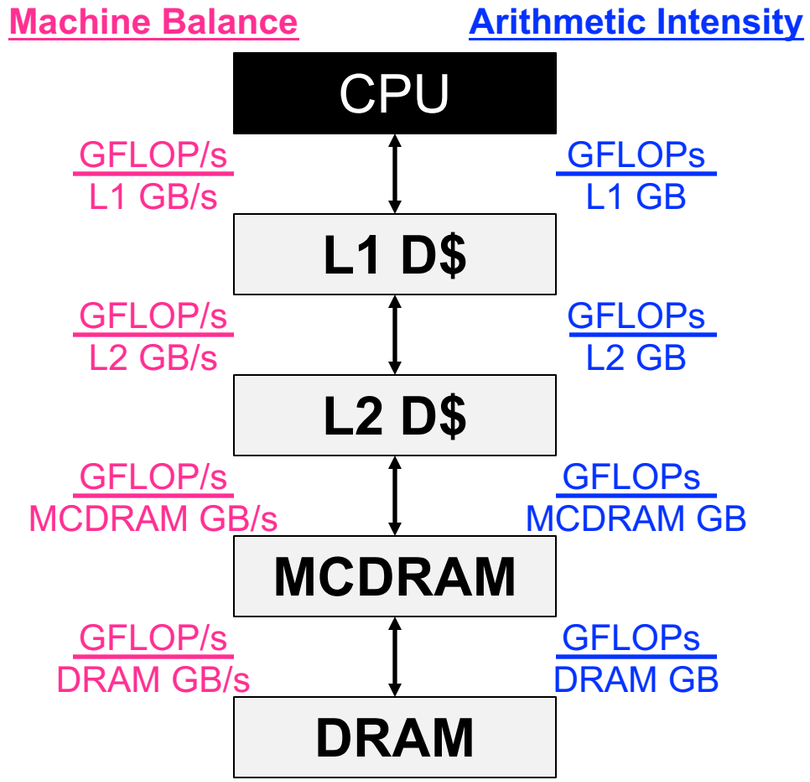
Hierarchical Roofline

- 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 unique peak and sustained bandwidths



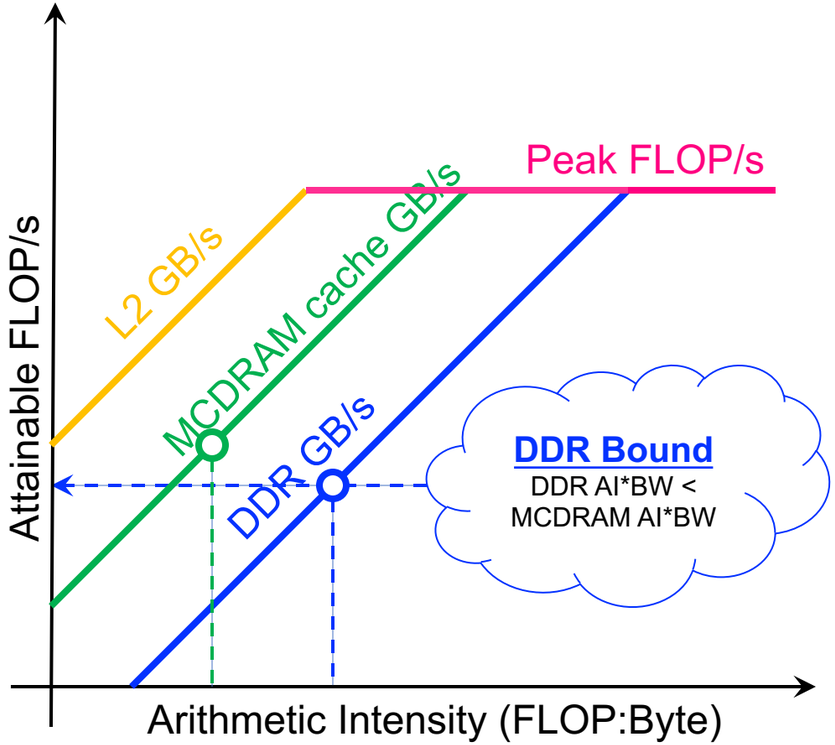
Hierarchical Roofline

- 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 unique peak and sustained bandwidths



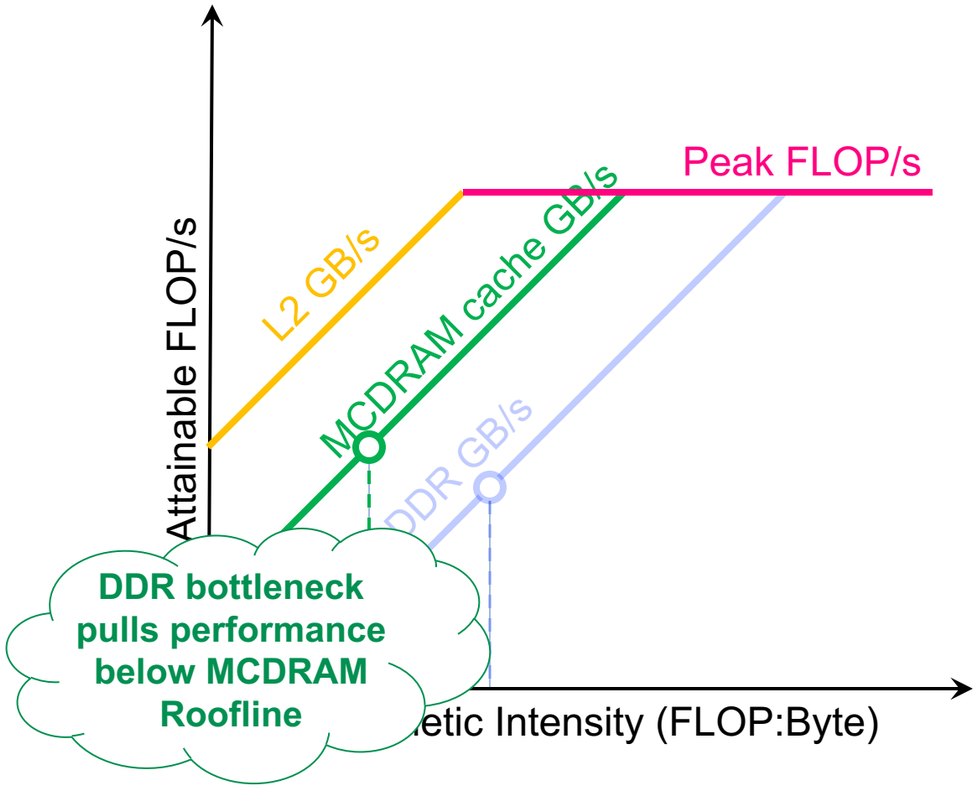
Hierarchical Roofline

- 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**



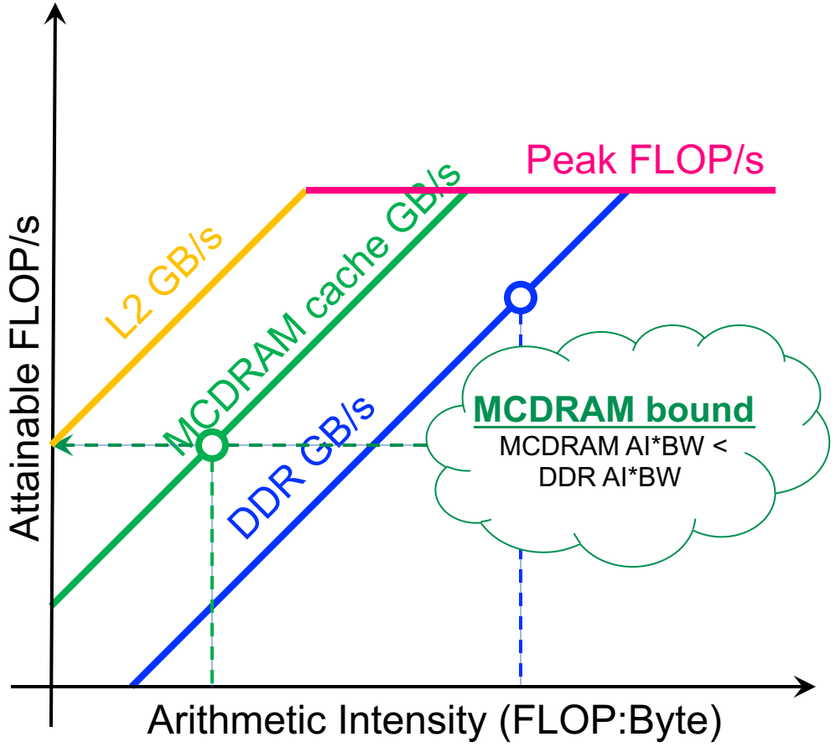
Hierarchical Roofline

- 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**



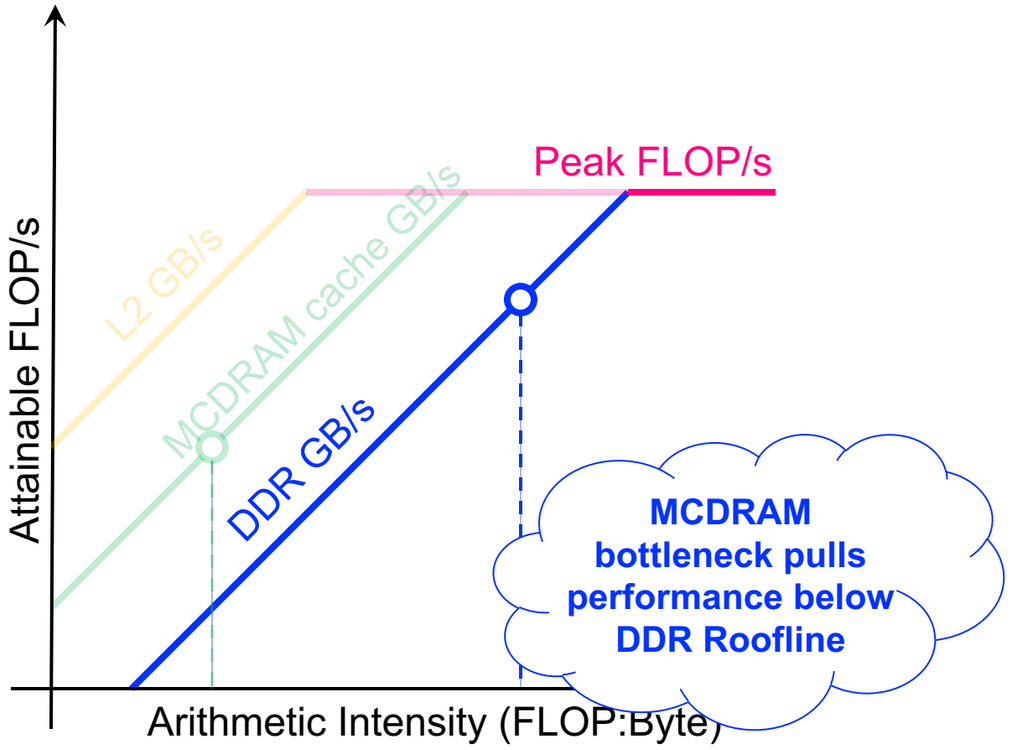
Hierarchical Roofline

- 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**



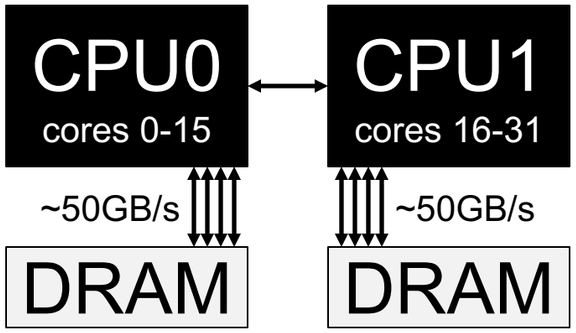
Hierarchical Roofline

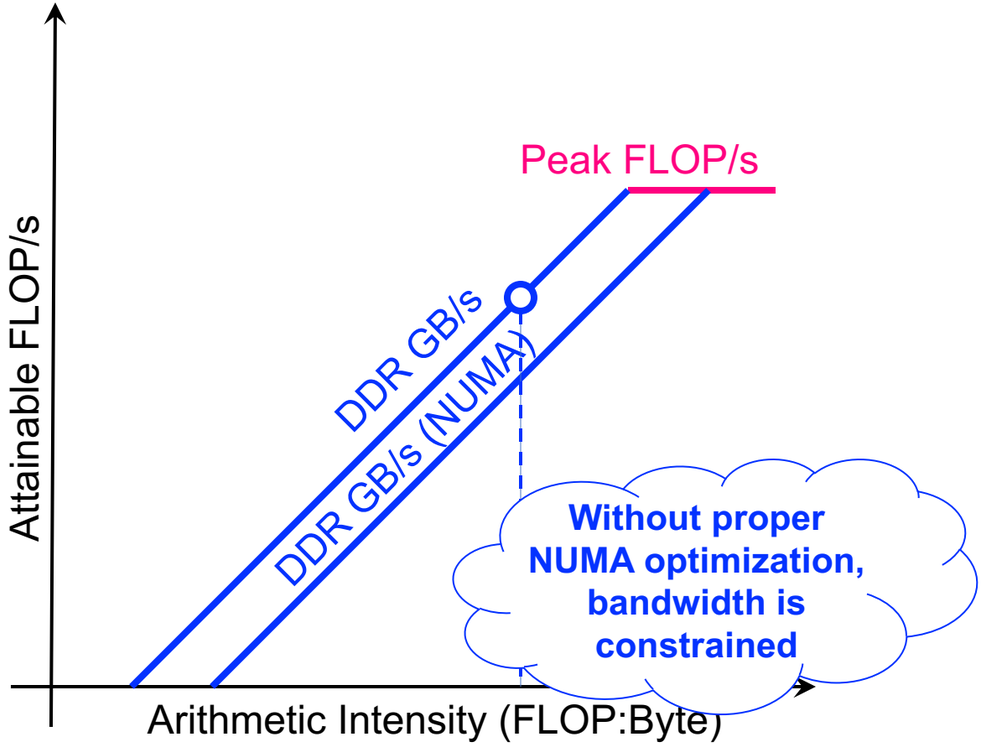
- Construct superposition of Rooflines…
  - Measure bandwidth
  - Measure AI for each level of memory
  - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, … DRAM)…
  - **… performance is bound by the minimum**



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





Extending the Roofline: In-Core Effects

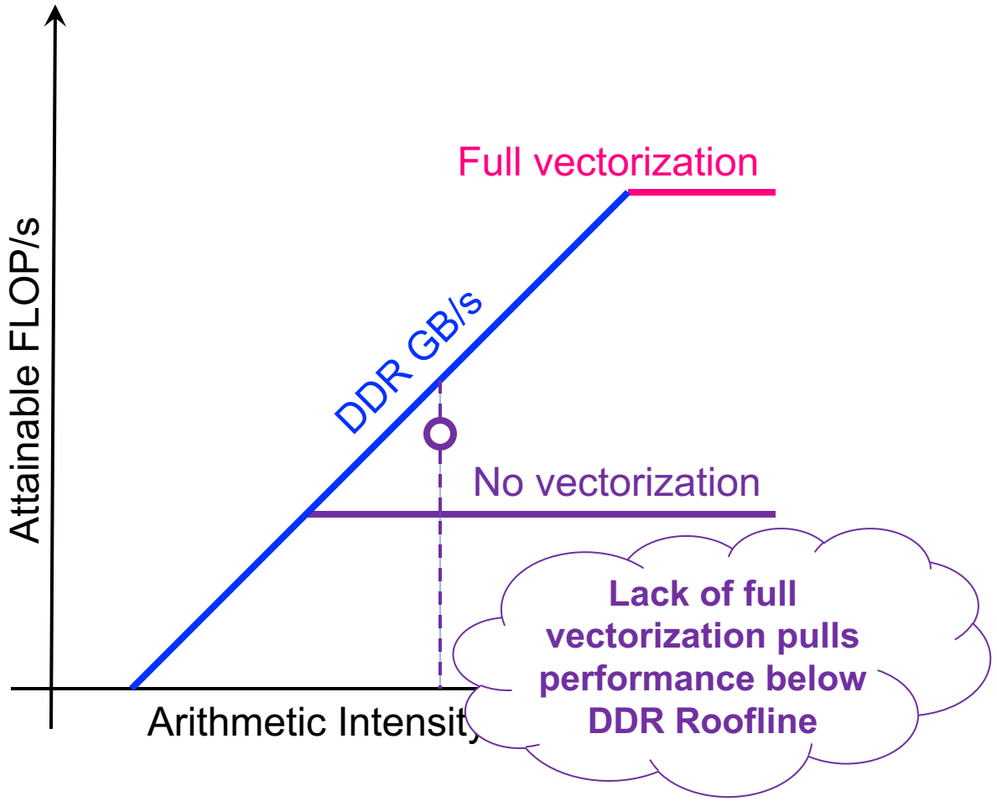
In-Core Parallelism

- We have assumed one can attain peak flops with high locality.
- In reality, we must …
  - Vectorize loops (16 flops per instruction)
  - Use special instructions (e.g. FMA)
  - Ensure FP instructions dominate the instruction mix
  - Use all cores & sockets

- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound

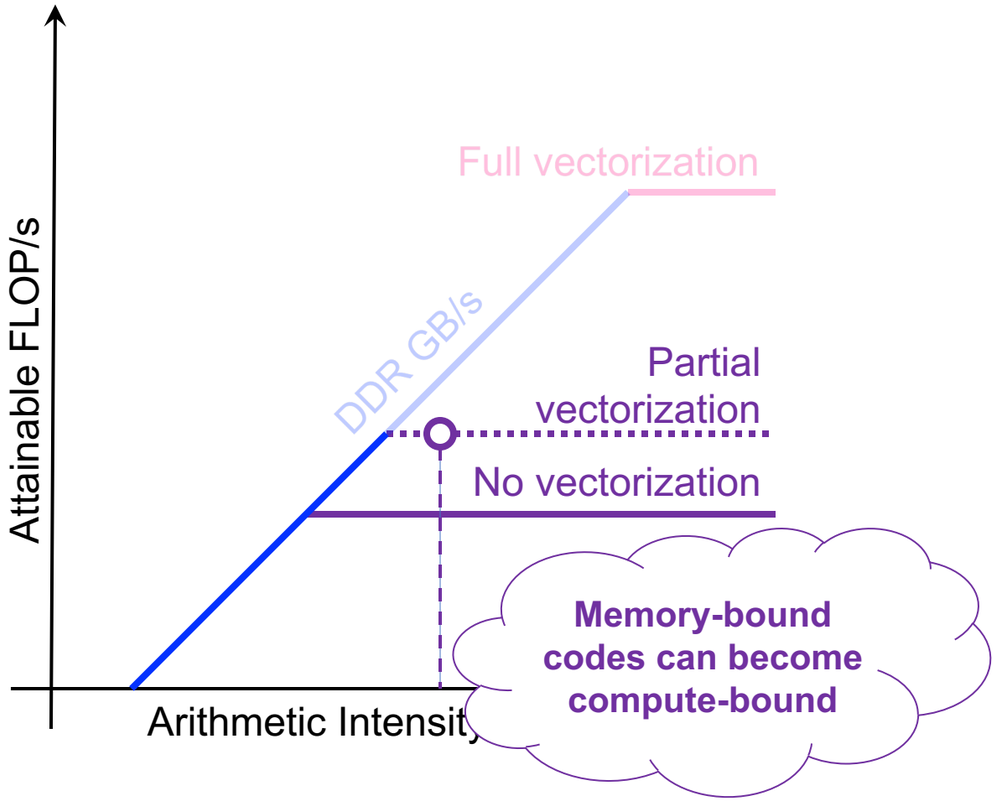
Data Parallelism (e.g. SIMD)

- Most processors exploit some form of SIMD or vectors.
  - KNL uses 512b vectors (8x64b)
  - GPUs use 32-thread warps (32x64b)
- In reality, applications are a mix of scalar and vector instructions.
  - **Performance is a weighted average between SIMD and no SIMD**



Vector instructions (e.g. SIMD)

- Most processors exploit some form of SIMD or vectors.
  - KNL uses 512b vectors (8x64b)
  - GPUs use 32-thread warps (32x64b)
- In reality, applications are a mix of scalar and vector instructions.
  - Performance is a weighted average between SIMD and no SIMD
  - ➢ **There is an implicit ceiling based on this weighted average**



Return of Complex Instruction Set Computing

- Death of Moore's Law is reinvigorating 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*
  - WMMA (Tensor Core): Z=AB+C …*Z,A,B,C are FP16 matrices*

➢ **Performance is now a weighted average of scalar, vector, FMA, and WMMA operations.**

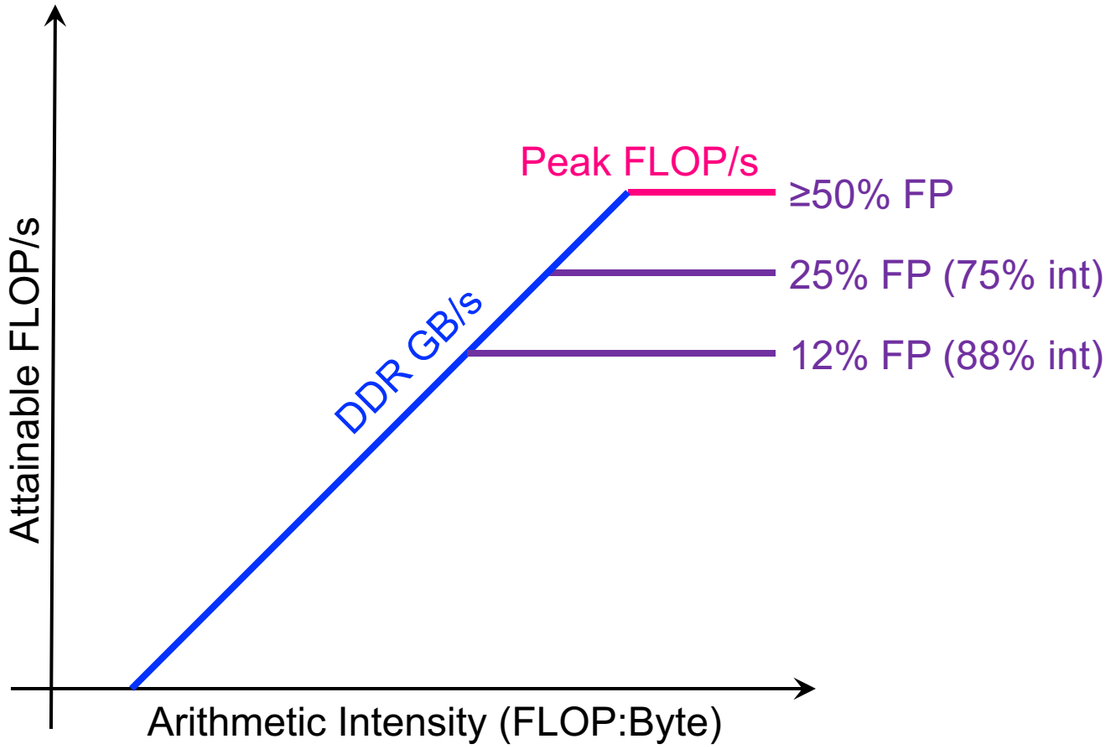
Superscalar vs. Instruction mix

- Superscalar processors have finite instruction fetch/decode/issue bandwidth (**e.g. 4 instructions per cycle**)
- Moreover, the number of FP units dictates the FP issue rate required to hit peak (**e.g. 2 vector instructions per cycle**)

➢ **Ratio of these two rates is the minimum FP instruction fraction required to hit peak**

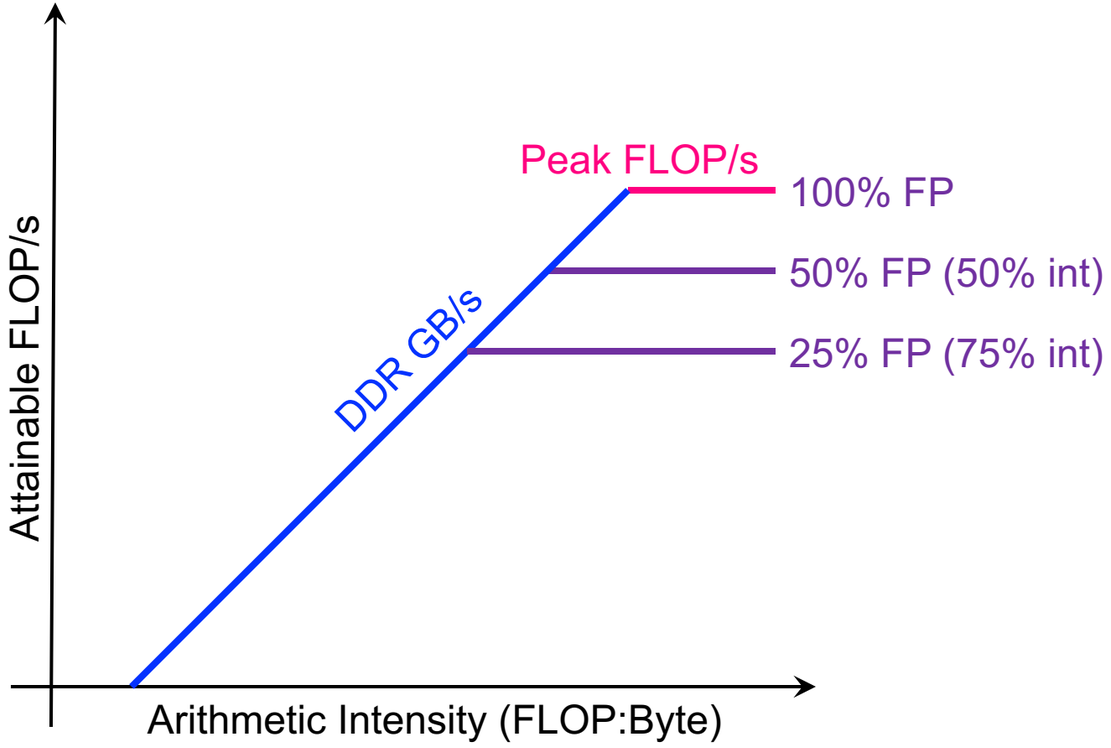
Superscalar vs. Instruction mix

- Haswell CPU
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance



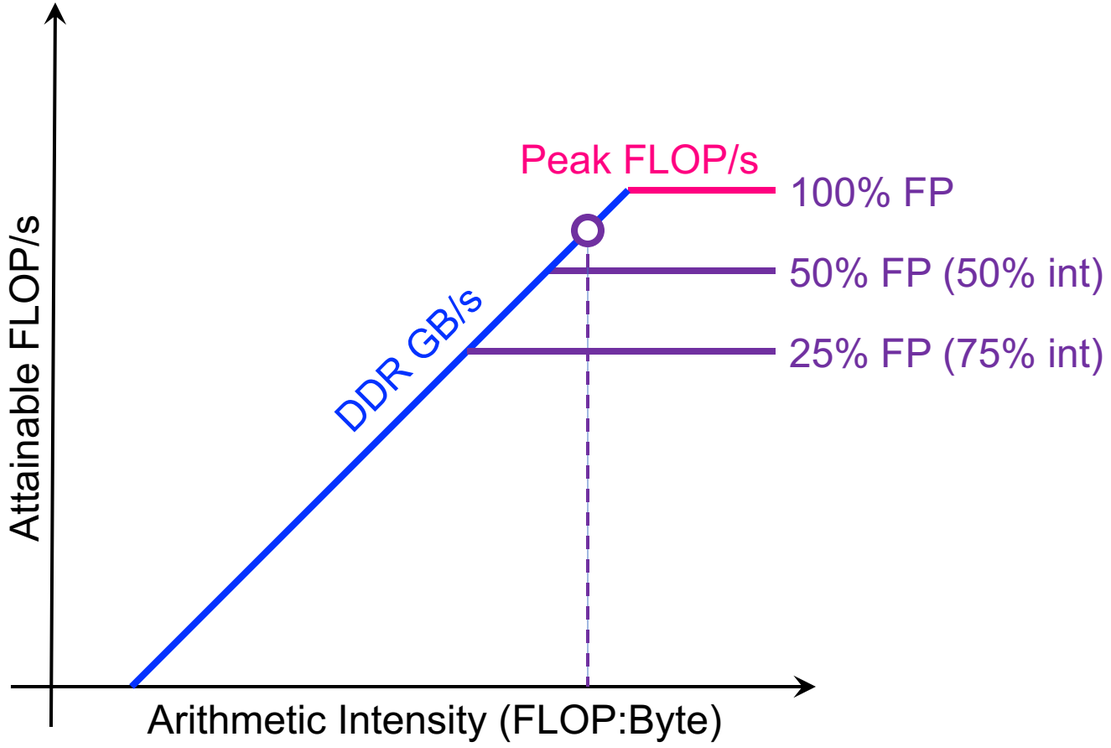
Superscalar vs. Instruction mix

- Haswell CPU
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- Conversely, on KNL…
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



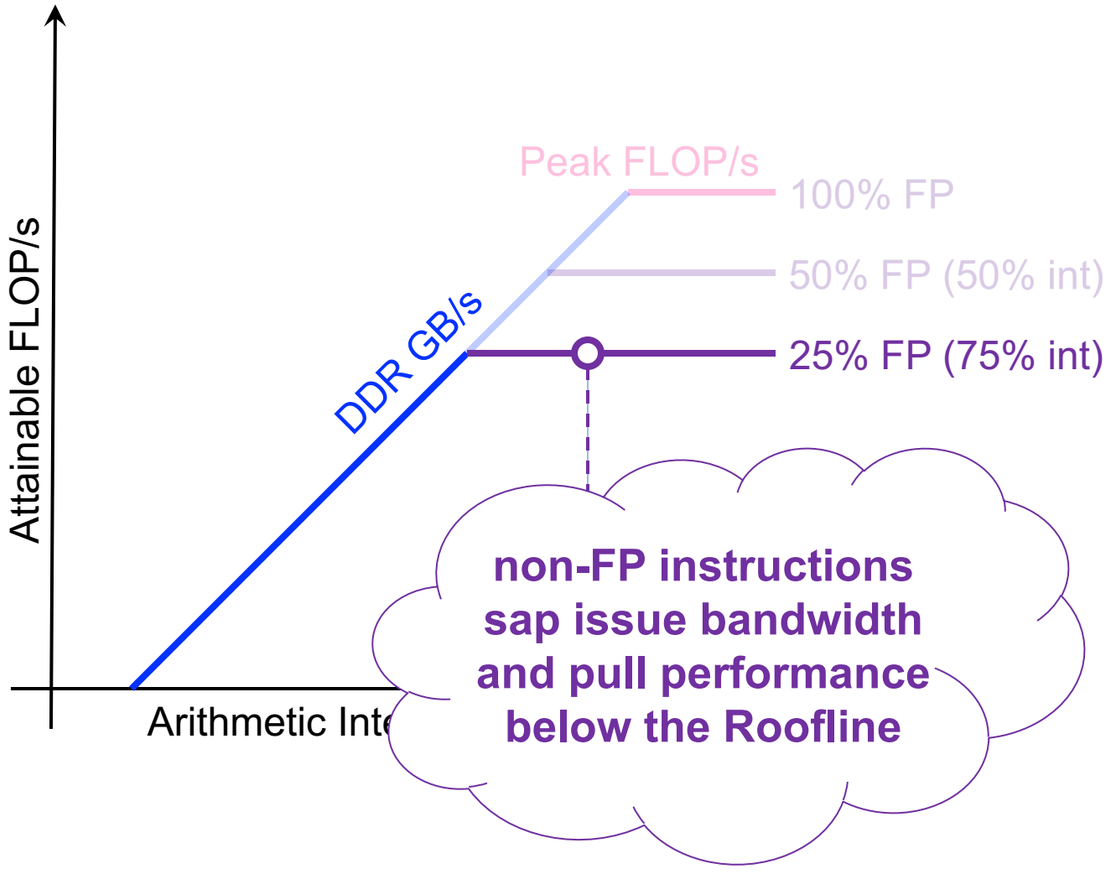
Superscalar vs. Instruction mix

- Haswell CPU
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- Conversely, on KNL…
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



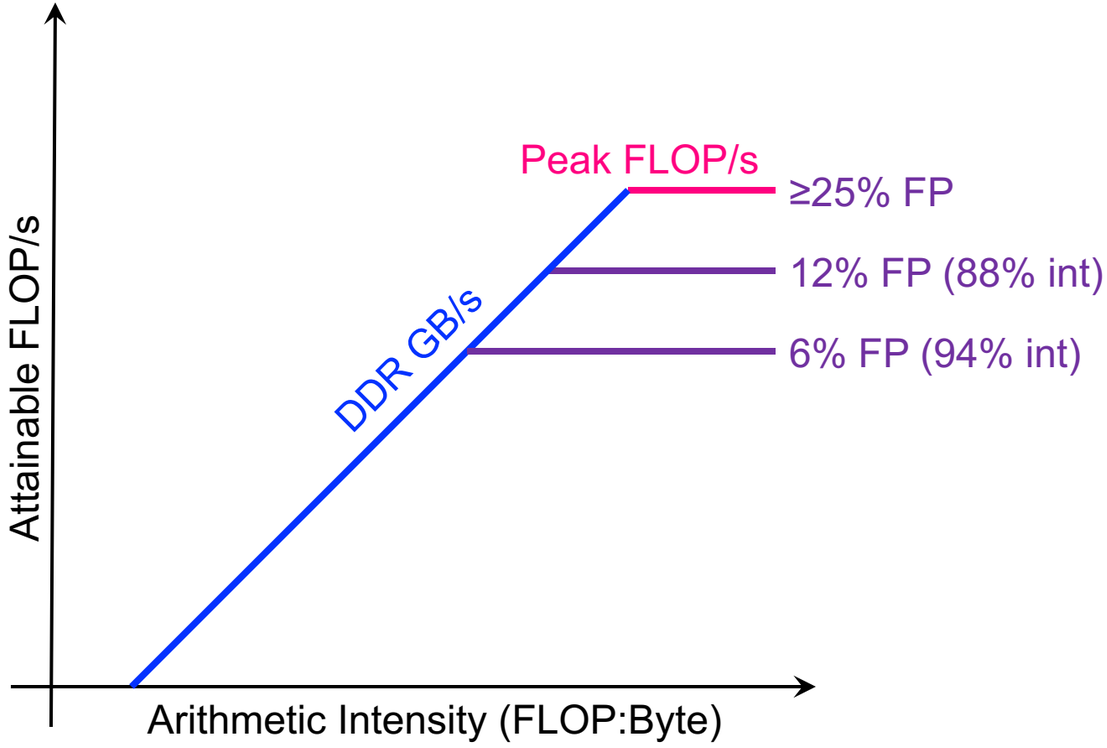
Superscalar vs. Instruction mix

- Haswell CPU
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- Conversely, on KNL…
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance
  - ➢ **Codes that would have been memory-bound are now decode/issue-bound.**



Superscalar vs. Instruction mix

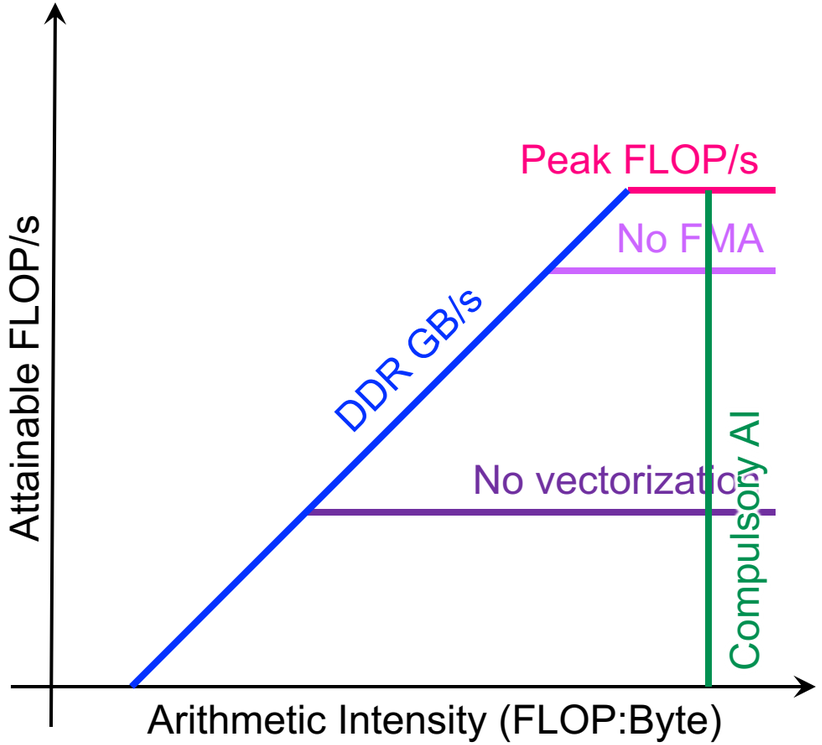
- On Volta, each SM is partitioned among 4 warp schedulers
- Each warp scheduler can dispatch 32 threads per cycle
- However, it can only execute 8 DP FP instructions per cycle.
- i.e. there is plenty of excess instruction issue bandwidth available for non-FP instructions.



Extending the Roofline: Modeling Cache Effects

Locality Walls

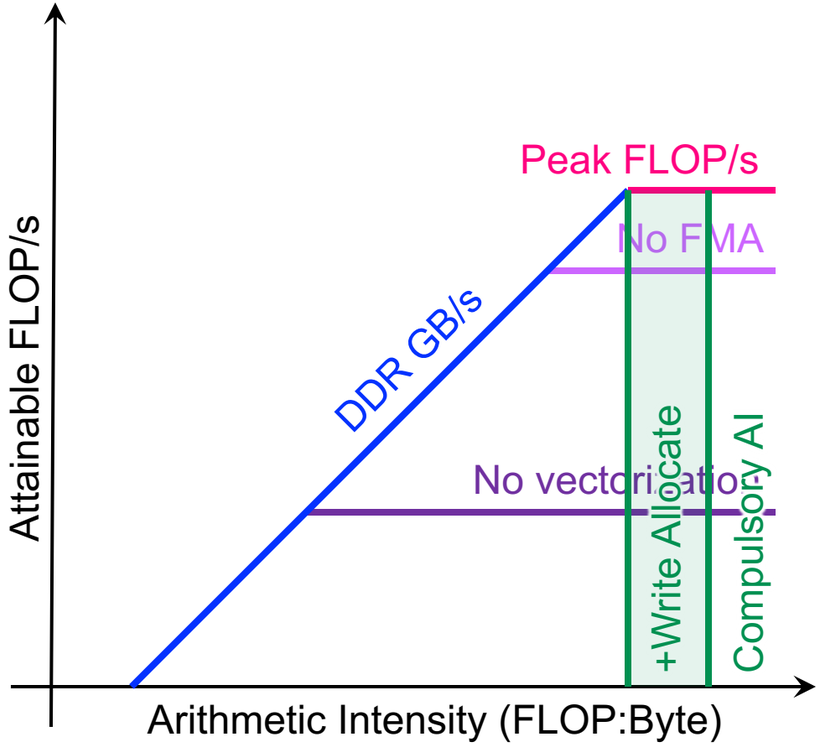
- Naively, we can bound AI using only compulsory cache misses



$$AI = \frac{\#FLOPs}{\text{Compulsory Misses}}$$

Locality Walls

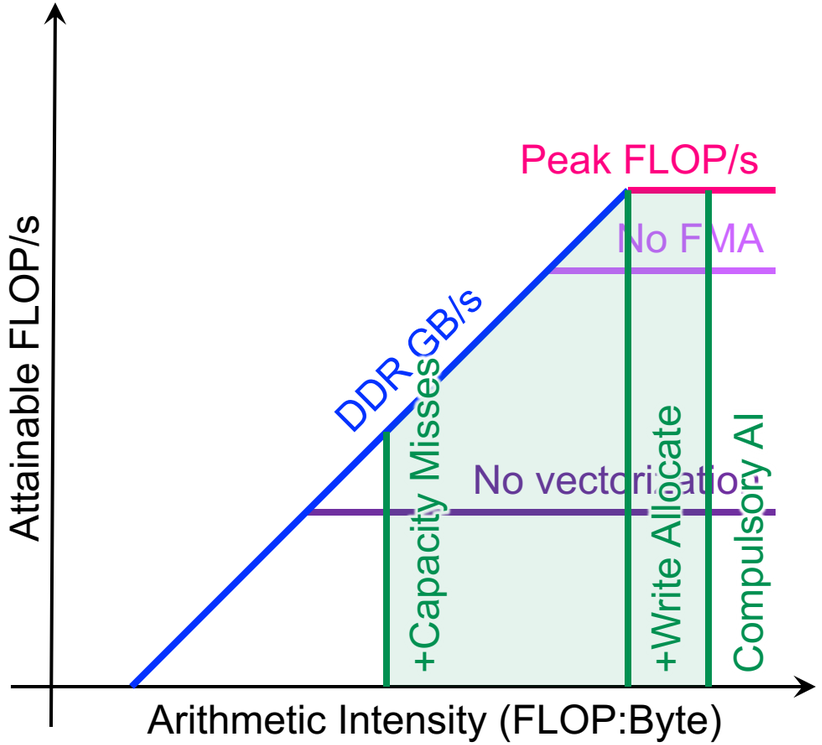
- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI



$$AI = \frac{\#FLOPs}{\text{Compulsory Misses + Write Allocates}}$$

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

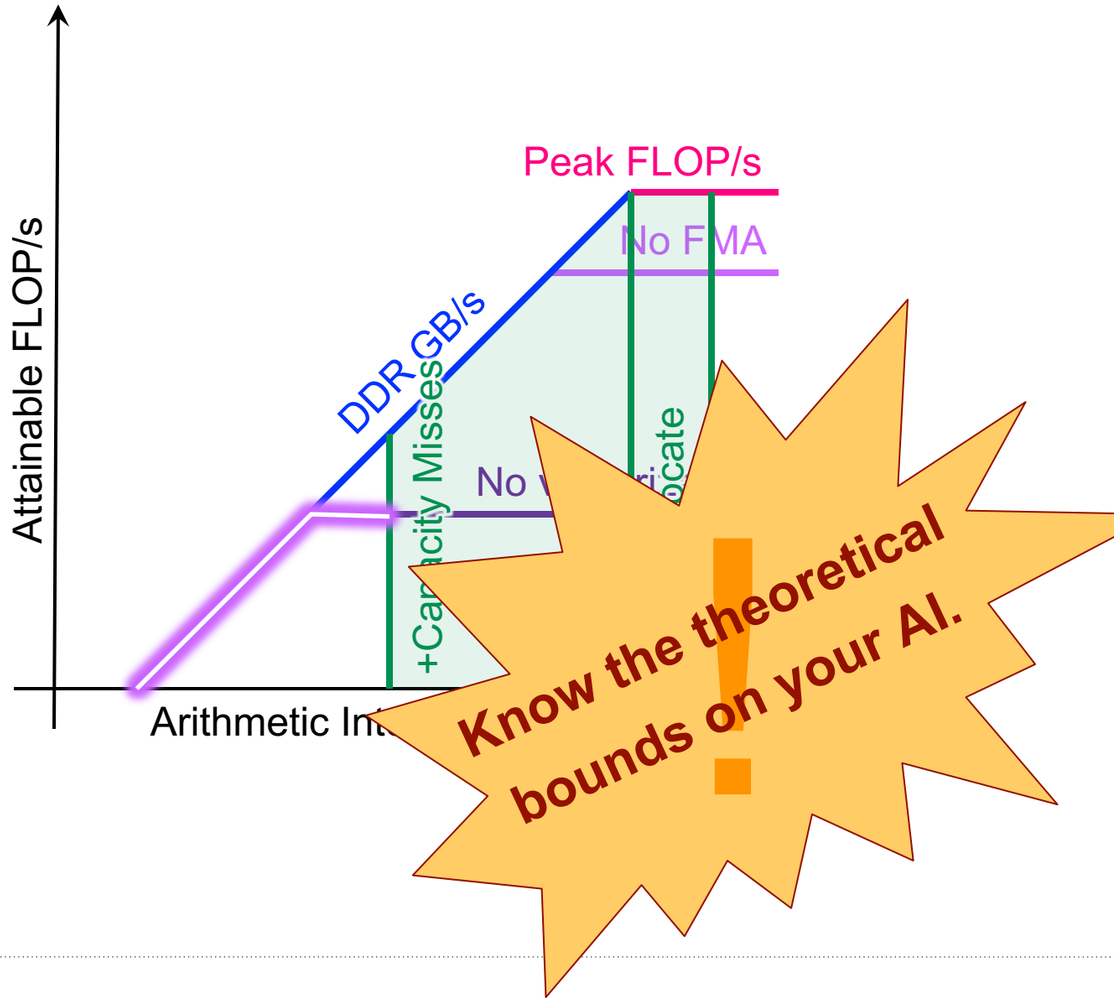


$$AI = \frac{\#FLOPs}{\text{Compulsory Misses + Write Allocates + Capacity Misses}}$$

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**

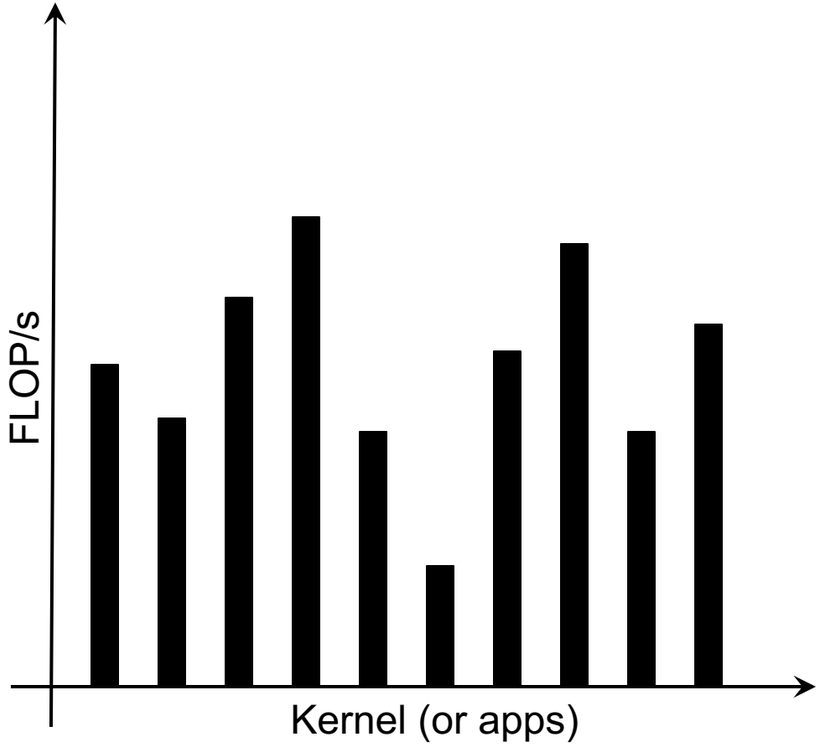
$$AI = \frac{\#FLOPs}{\text{Compulsory Misses} + \text{Write Allocates} + \text{Capacity Misses}}$$



So Why is Roofline Useful?

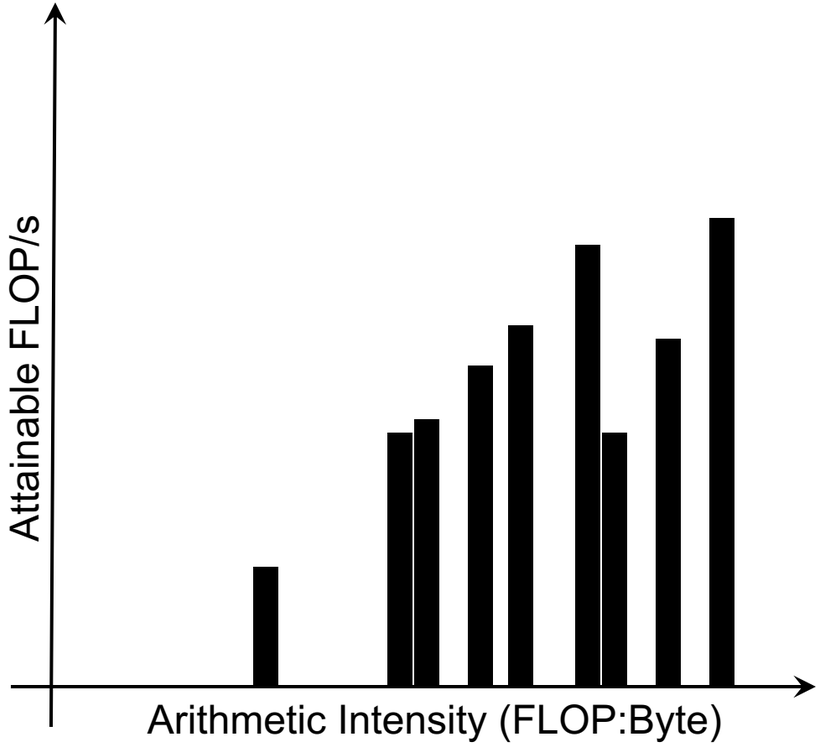
Why is Roofline Useful?

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



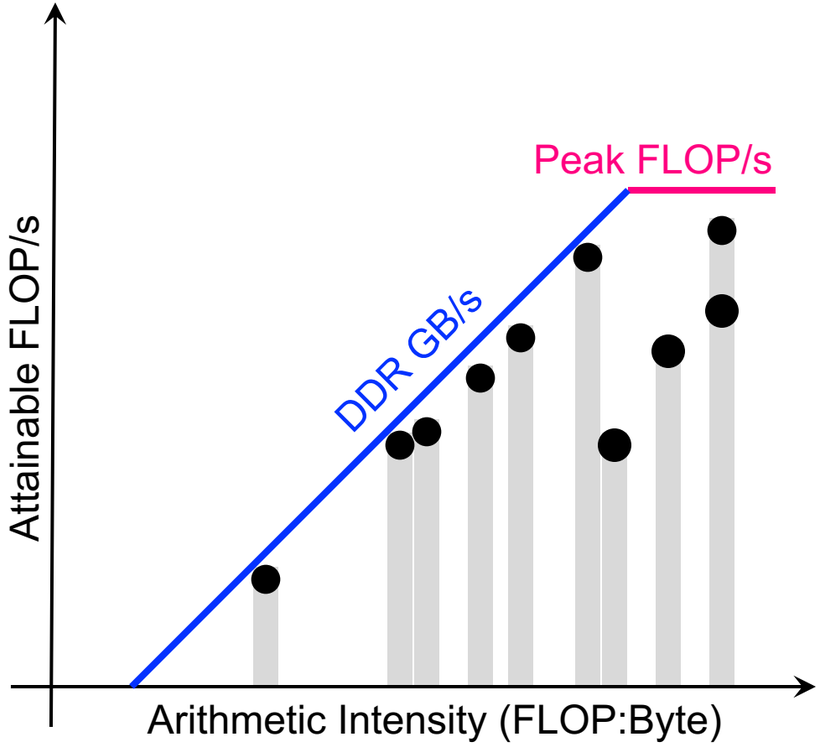
Why is Roofline Useful?

- We can sort kernels by AI …



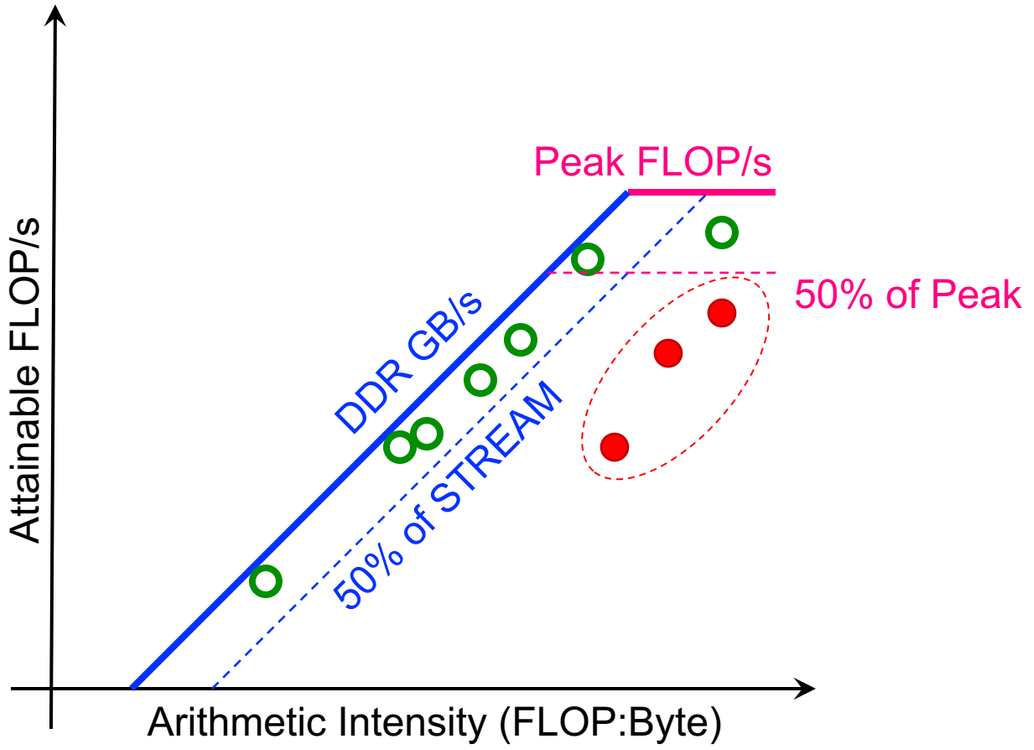
Why is Roofline Useful?

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



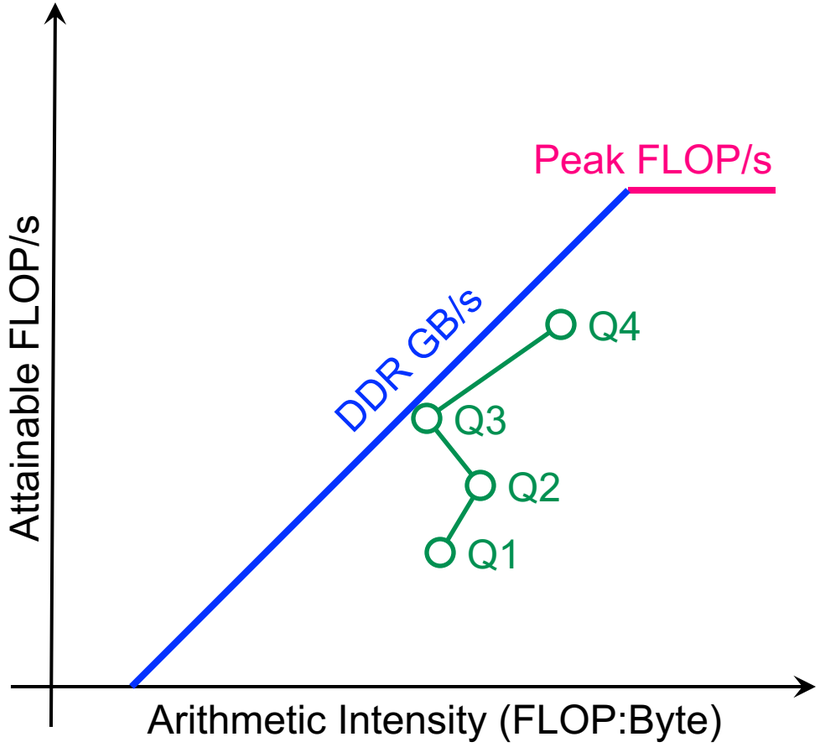
Why is Roofline Useful?

- 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



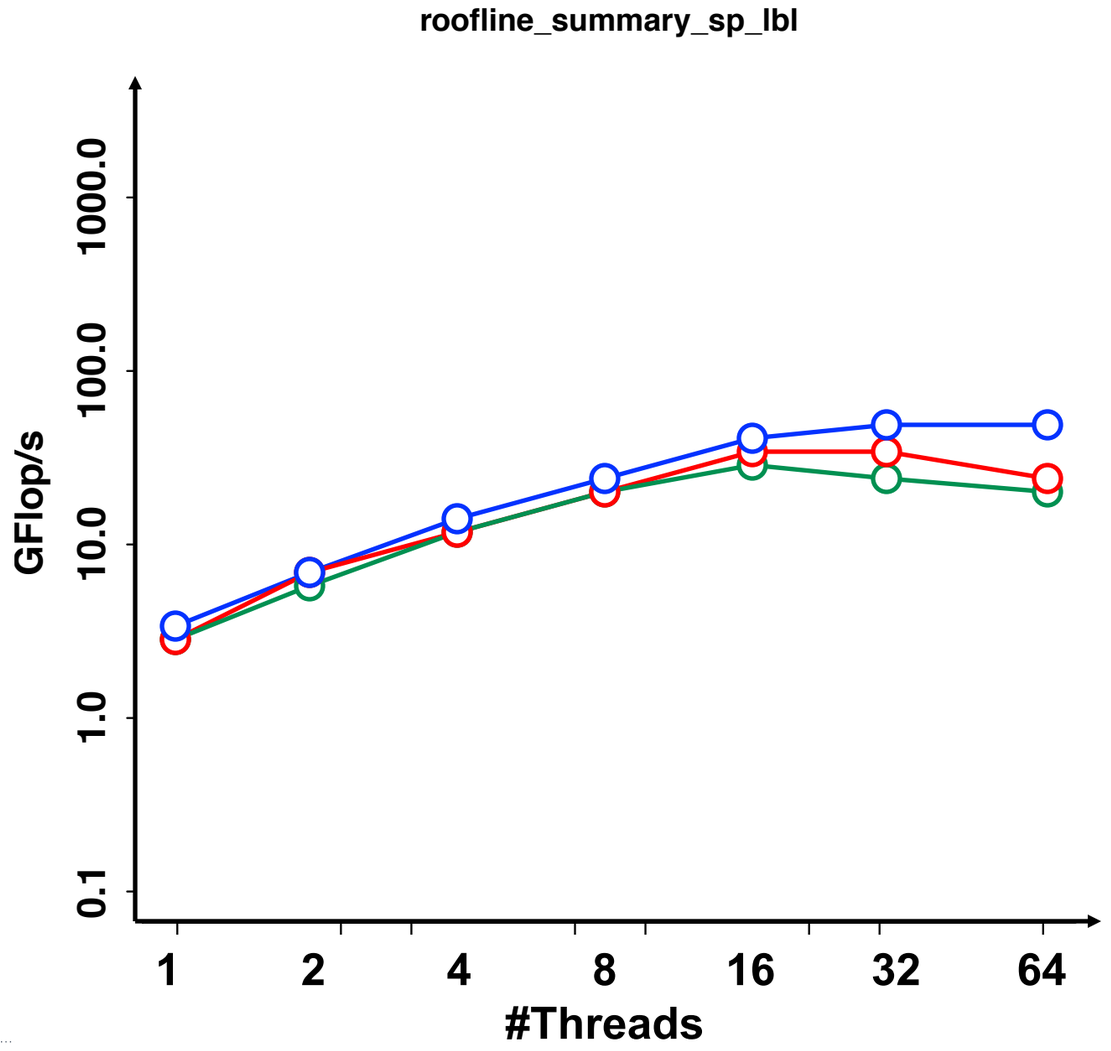
Tracking Progress Towards Optimality

- One can conduct a Roofline optimization after every optimization (or once per quarter)
  - Tracks progress towards optimality
  - Allows one to quantitatively speak to ultimate performance / KPPs
  - Can be used as a motivator for new algorithms.



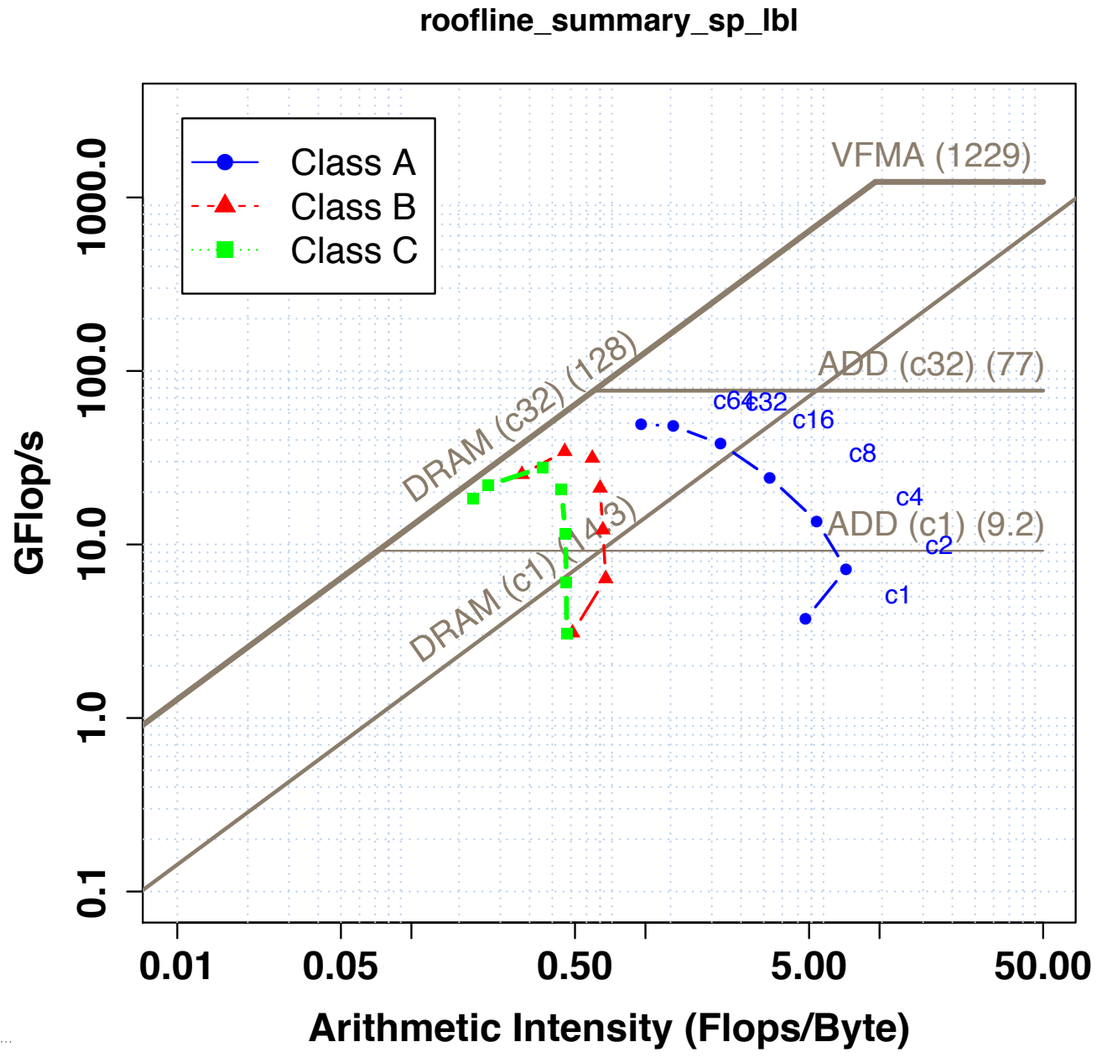
Roofline Scaling Trajectories

- Often, one plots performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable information.



Roofline Scaling Trajectories

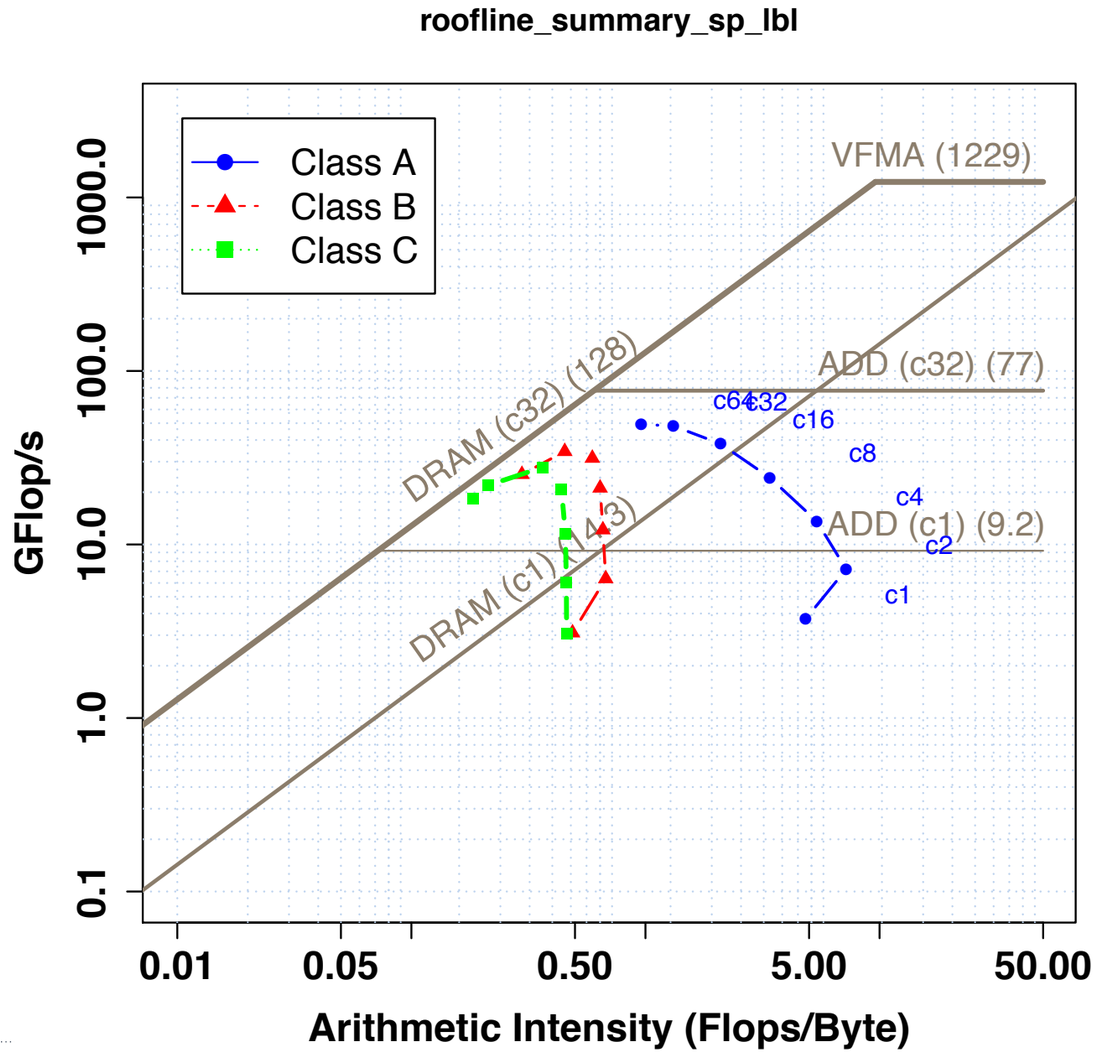
- Often, one plots performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable information.
- Khaled Ibrahim developed a new way of using Roofline to analyze thread (or process) scalability
  - Create a 2D scatter plot of performance as a function of AI and thread concurrency
  - Can identify loss in performance due to increased cache pressure



Khaled Ibrahim, Samuel Williams, Leonid Oliker, "Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis", HPCS Special Session on High Performance Computing Benchmarking and Optimization (HPBench), July 2018.

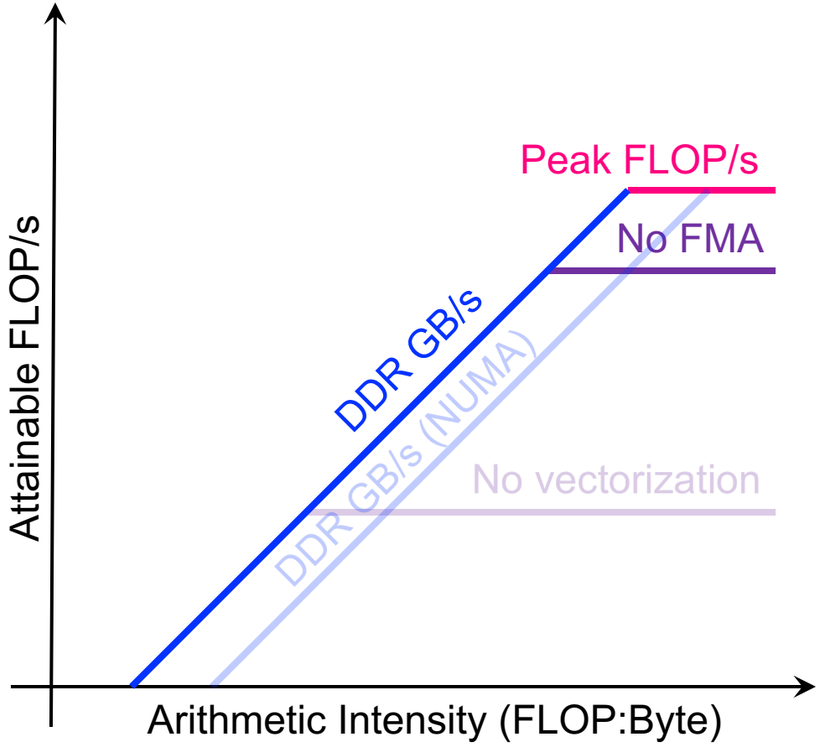
Roofline Scaling Trajectories

- Observe…
  - AI (data movement) varies with both thread concurrency and problem size
  - Large problems (green and red) move much more data per thread, and eventually exhaust cache capacity
  - Resultant fall in AI means they hit the bandwidth ceiling quickly and degrade.
  - Smaller problems see reduced AI, but don't hit the bandwidth ceiling



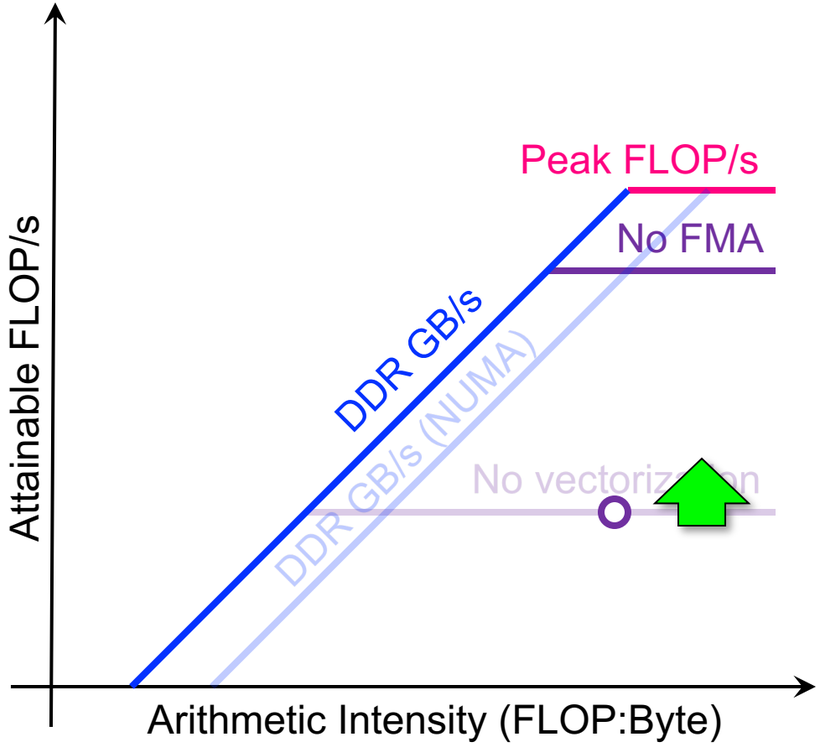
Driving Performance Optimization

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



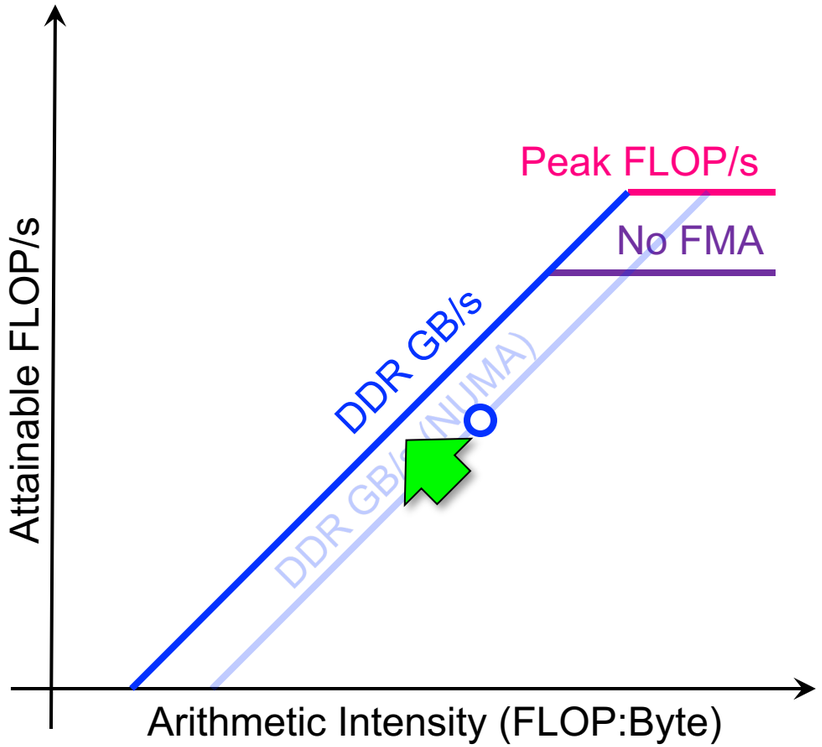
Driving Performance Optimization

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



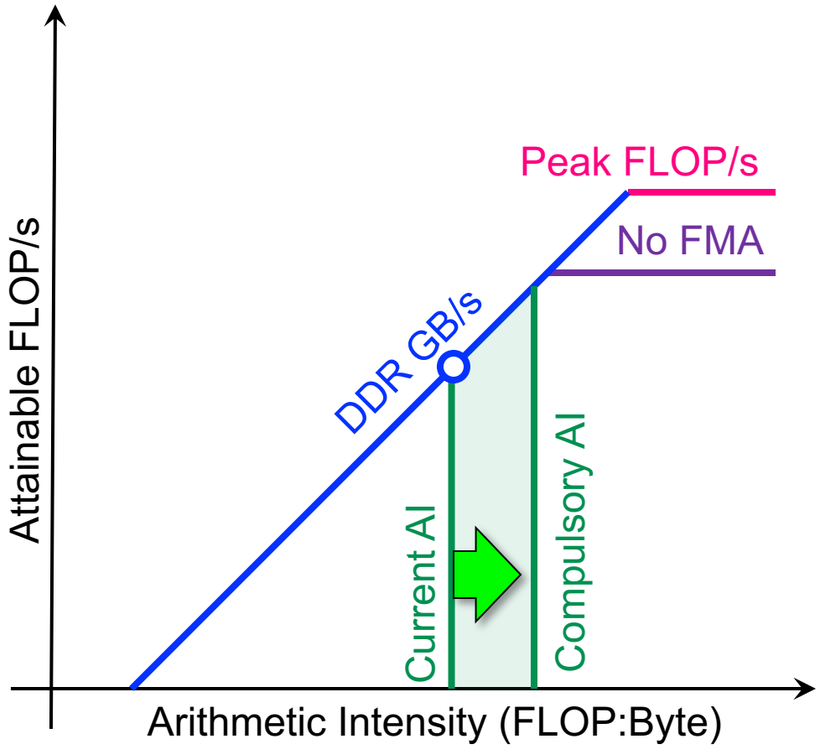
Driving Performance Optimization

- 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)**



Driving 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)**



How do I build and use Roofline?

See also:
https://docs.nersc.gov/programming/performance-debugging-tools/roofline/

Machine Characterization

- **"Theoretical Performance"** numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode at low concurrency
  - Underclocking for AVX
  - Compiler failing on high-AI loops.

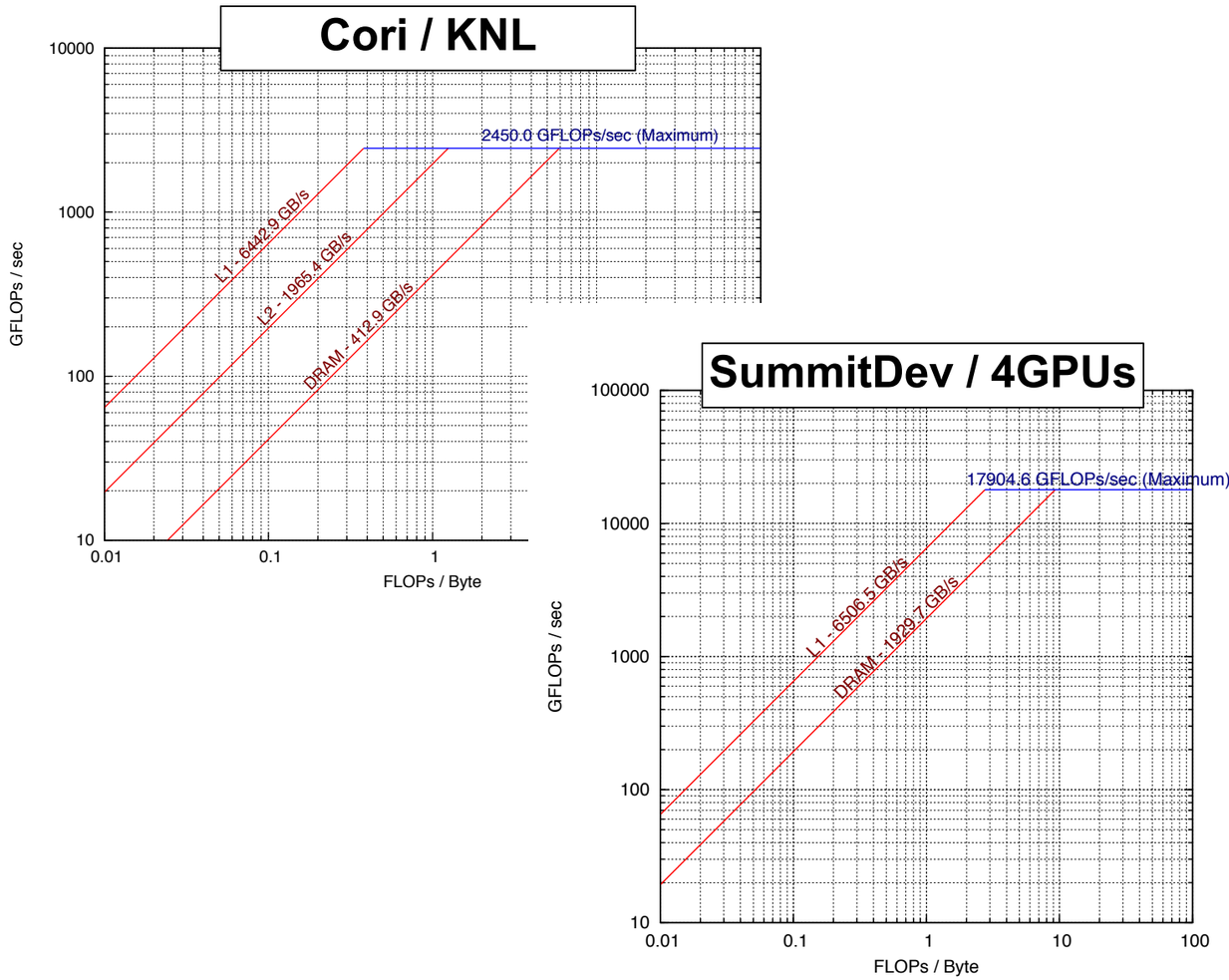
➢ **Take marketing numbers with a grain of salt**

Machine Characterization

- To create a Roofline model, we must benchmark…
  - **Sustained Flops**
    - Double/single/half precision
    - With and without FMA (e.g. compiler flag)
    - With and without SIMD (e.g. compiler flag)
  - **Sustained Bandwidth**
    - Measure between each level of memory/cache
    - Iterate on working sets of various sizes and identify plateaus
    - Identify bandwidth asymmetry (read:write ratio)
- Benchmark must run long enough to observe effects of power throttling

Machine Characterization

- **"Theoretical Performance"** numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode / Underclock for AVX
  - compiler failings on high-AI loops.
- LBNL 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://bitbucket.org/berkeleylab/cs-roofline-toolkit/

https://github.com/cyanguwa/nersc-roofline/

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

Measuring Application AI and Performance

- To characterize execution with Roofline we need…
  - **Time**
  - **Flops** (=> FLOPs / time)
  - **Data movement** between each level of memory (=> FLOPs / 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 FLOPs?

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

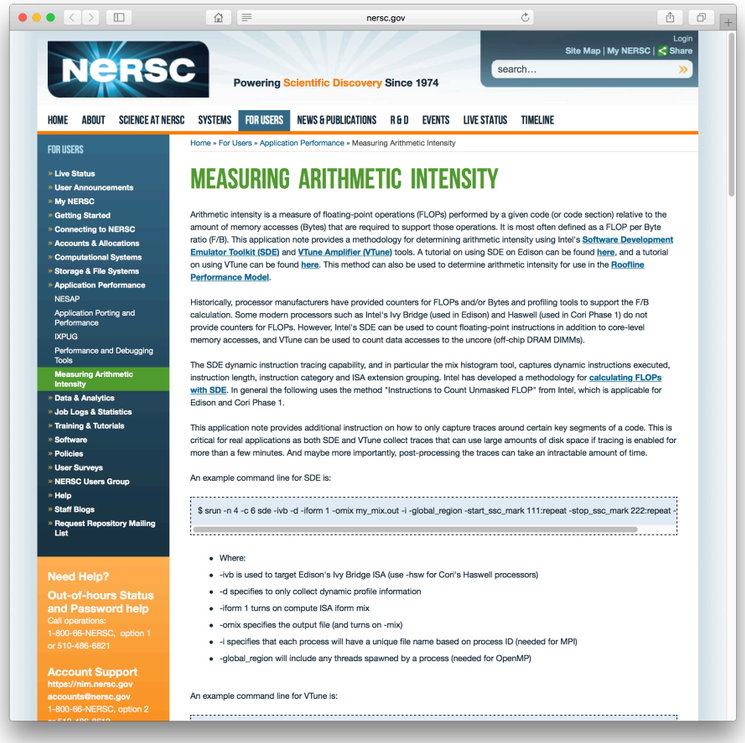
✗ >10x overhead (short runs / reduced concurrency)

Initially 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 FLOPs (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/

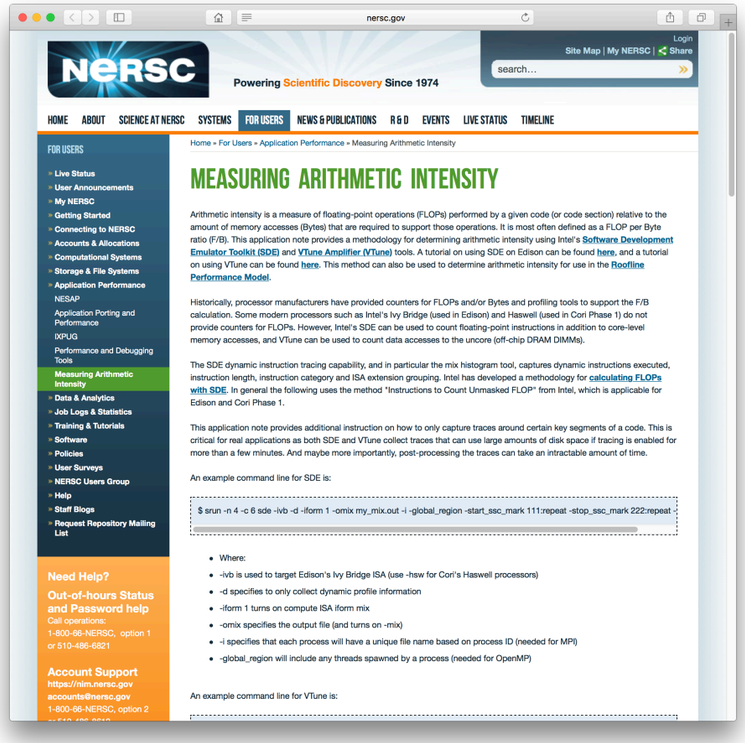
NERSC is LBL's production computing division
CRD is LBL's Computational Research Division
NESAP is NERSC's KNL application readiness project
LBL is part of SUPER (DOE SciDAC3 Computer Science Institute)

More Recently…

- 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 **LIKWID** performance counter tool (NERSC/Cray approved) to access uncore counters

➢ Accurate measurement of FLOPs (HSW) and DRAM data movement (HSW and KNL)

➢ Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori…

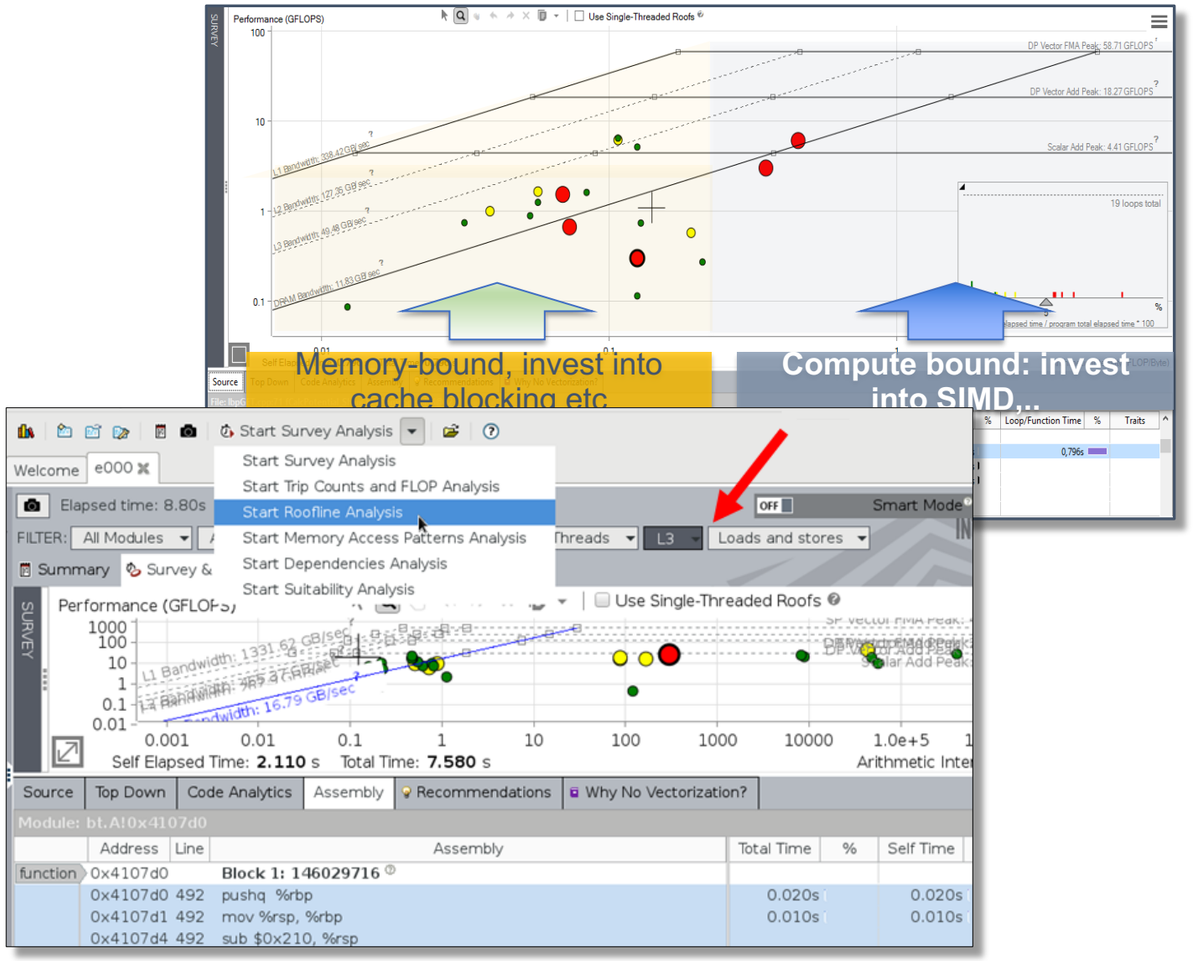


http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

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



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

Advisor on NERSC's Cori

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

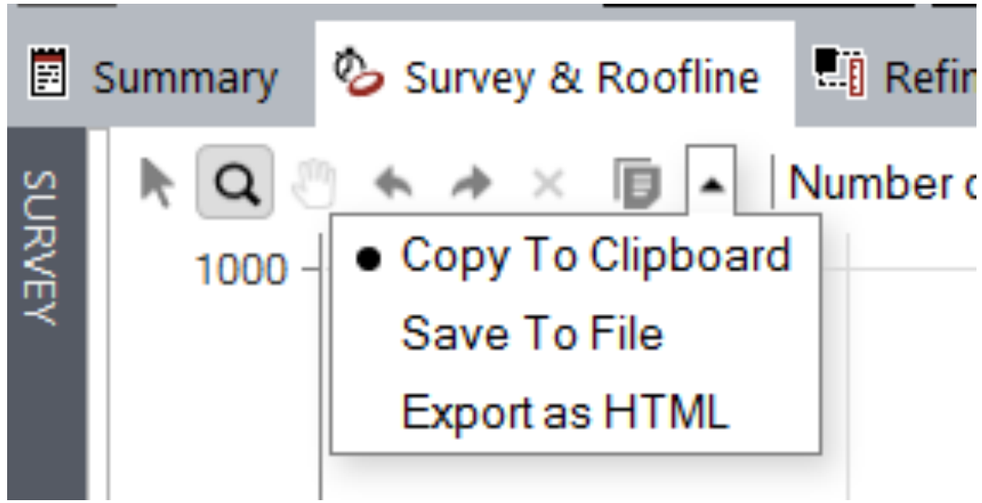
```
•module load advisor/2018.integrated_roofline
•cc -g -dynamic -openmp -O2 -o mycode.exe mycode.c
```

- Best to run advisor only on rank 0... srun calls a script like…

```
•#!/bin/bash
•if [[ $SLURM_PROCID == 0 ]];then
•advixe-cl -collect=survey --project-dir knl-result -data-limit=0 -- ./a.out
•else
•sleep 30
•./a.out
•fi
```

Exporting Roofline Figures

- Advisor can directly export a
- HTML Roofline figure …



- Alternately, you can output directly from the command line (no GUI needed)…

```
advixe-cl -report roofline --project-dir ./your_project > roofline.html
```

Summary

Summary

- Performance Models
- Roofline Model
- Tools for Roofline Analysis…
  - Machine Characterization (ERT)
  - Using LIKWID to access performance counters
  - Using SDE to get more accurate FLOP counts
  - Using Advisor to provide a single tool that integrates cache simulation and accurate FLOP counts.
  - Using NVProf to affect Roofline on GPUs

Questions?

Backup

Tools for Roofline Analysis on GPUs

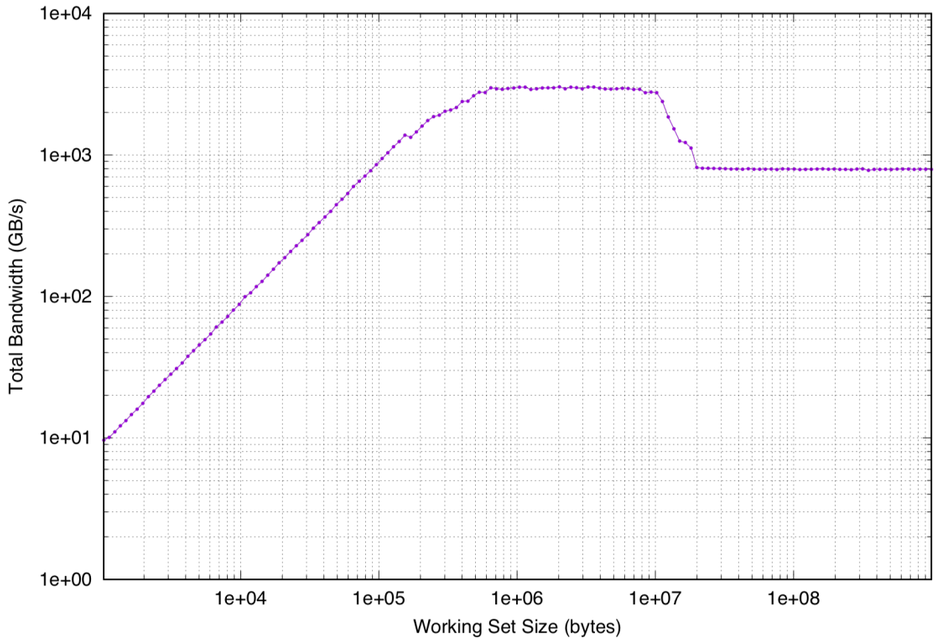
slides provided by Charlene Yang (CJYang@lbl.gov)

Roofline on GPUs (Overview)

- Use ERT to obtain empirical Roofline ceilings
  - compute: FMA, no-FMA
  - bandwidth: system memory, device memory, L2, L1
- Use nvprof to obtain application performance
  - FLOPs: active non-predicated threads, divides-aware
  - bytes: read + write; system memory, device memory, L2, L1
  - runtime: --print-gpu-summary, --print-gpu-trace
- Plot Roofline with Python and Matplotlib

Characterizing NVIDIA GPUs

- Empirical Roofline Toolkit (ERT)
- https://bitbucket.org/berkeleylab/cs-roofline-toolkit/
- Sweeps through a variety of configurations:
  - 1 data element per thread -> multiple
  - 1 FLOP operation per data element -> multiple
  - number of threadblocks/threads
  - number of trails, dataset sizes, *etc*
- Four components
  - Driver.c, Kernel.c, configuration script, and job script



Characterizing GPU-accelerated Applications

- Three measurements: **Time, FLOPs, Bytes (on each cache level)**

$$\text{Performance} = \frac{nvprof\ \text{FLOPs}}{\text{Runtime}}, \quad \text{Arithmetic Intensity} = \frac{nvprof\ \text{FLOPs}}{nvprof\ \text{Data Movement}}$$

- Runtime:
  - time per invocation of a kernel
    `nvprof --print-gpu-trace ./application args`
  - average time over multiple invocations
    `nvprof --print-gpu-summary ./application args`
  - same kernel with different input parameters are grouped separately

Characterizing GPU-accelerated Applications

- FLOPs:
  - predication aware, and divides aware, **dp/dp_add/dp_mul/dp_fma**, **sp***
    nvprof --kernels ‘kernel_name’ --metrics ‘flop_count_xx’ ./application
- Bytes for different cache levels to construct hierarchical Roofline
    nvprof --kernels ‘kernel_name’ --metrics ‘metric_name’./application
  - Bytes = (read transactions + write transactions) x transaction size

Memory Level	Metrics	Transaction Size
L1 Cache	`gld_transactions, gst_transactions`	32B
L2 Cache	`l2_read_transactions, l2_write_transactions`	32B
Device Memory	`dram_read_transactions, dram_write_transactions`	32B
System Memory	`system_read_transactions, system_write_transactions`	32B

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 sysmem_read_bytes --metrics sysmem_write_bytes ./backup-bin/hpgmg-fv-fp 5 8`

- Can collect all metrics at once or one at a time (slowdown)
- Output in CSV; Python/Excel for multiple output files

```
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                         sysmem_read_bytes                   System Memory Read Bytes           0           0           0
          1                        sysmem_write_bytes                  System Memory Write Bytes         160         160         160
```

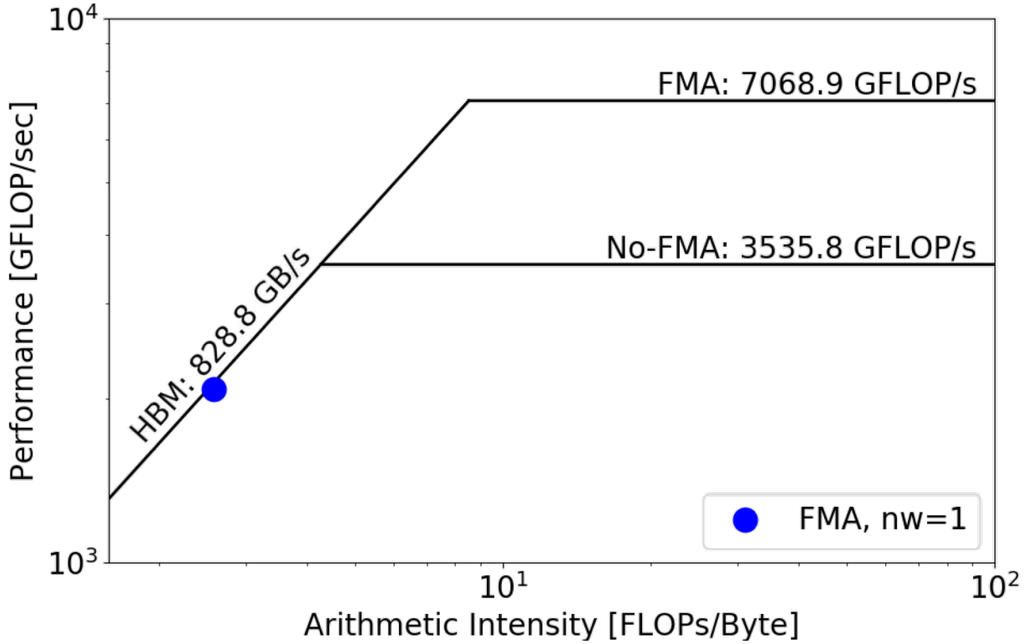
Plotting Rooflines of NVProf Data

- Python scripts using Matplotlib
  https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting
- Simple example: **`plot_roofline.py data.txt`**
- Tweaking needed for more sophisticated plotting, see examples

data.txt

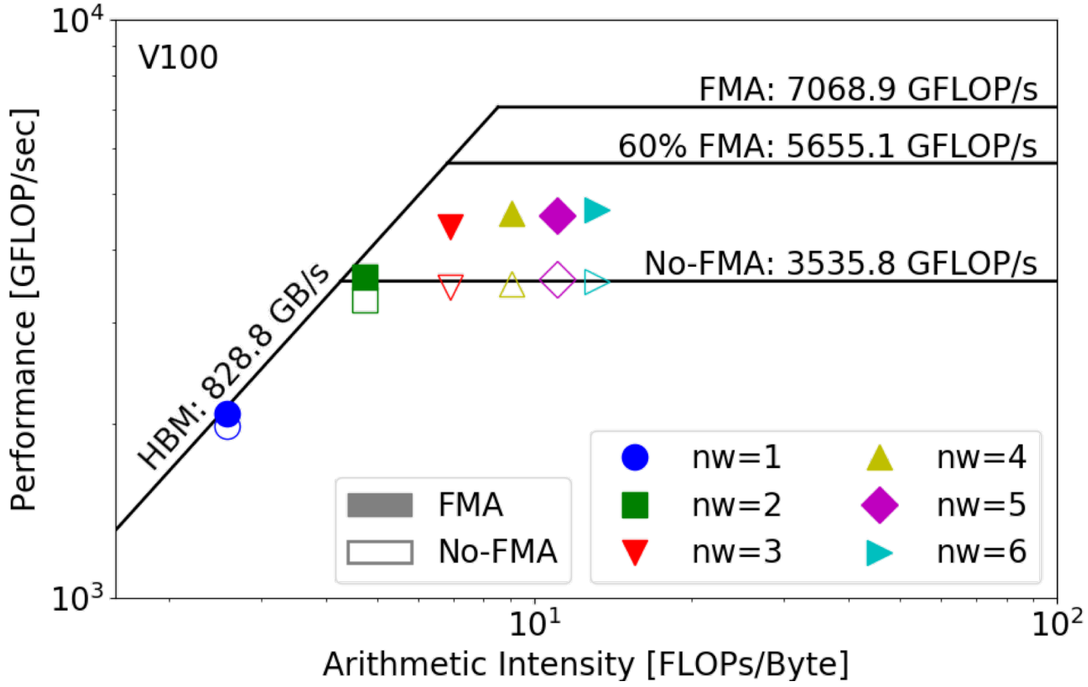
```
# all data is space delimited
memroofs 828.758
mem_roof_names 'HBM'
comproofs 7068.86 3535.79
comp_roof_names 'FMA' 'No-FMA'

# omit the following if only plotting roofs
# AI: arithmetic intensity; GFLOPs: performance
AI 2.584785579
GFLOPs 2085.756683
labels 'FMA, nw=1'
```



HBM Roofline on GPUs

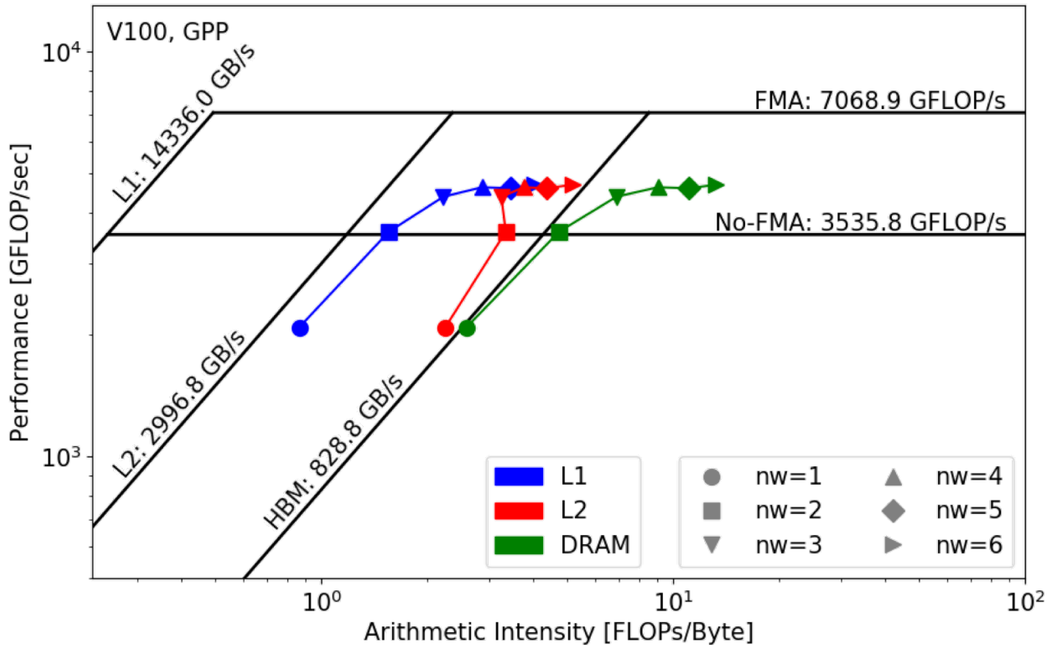
- Use BerkeleyGW Proxy app GPP to see GPU effects
- HBM Roofline
- AI increases as **`nw`** grows
- bandwidth bound → compute bound
- Disable FMA in the compiler…
  - (**`-fmad=true/false`**)
  - "No-FMA" converges to its ceiling
  - But FMA doesn't



Hierarchical Roofline on GPUs

- GPP is HBM bound
- L1/L2 performance far from L1/L2 ceiling

- FLOPs are proportional to **`nw`**
- Increase in HBM AI → **HBM bytes approx. constant (good L2 locality)**
- Slow increase in L2 AI → **L2 bytes increase for `nw>1` (poor L1 locality)**
- Increase in L1 AI → **L1 bytes approx. constant (good register file locality)**



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 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]Technology Preview, not in official product roadmap so far.

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

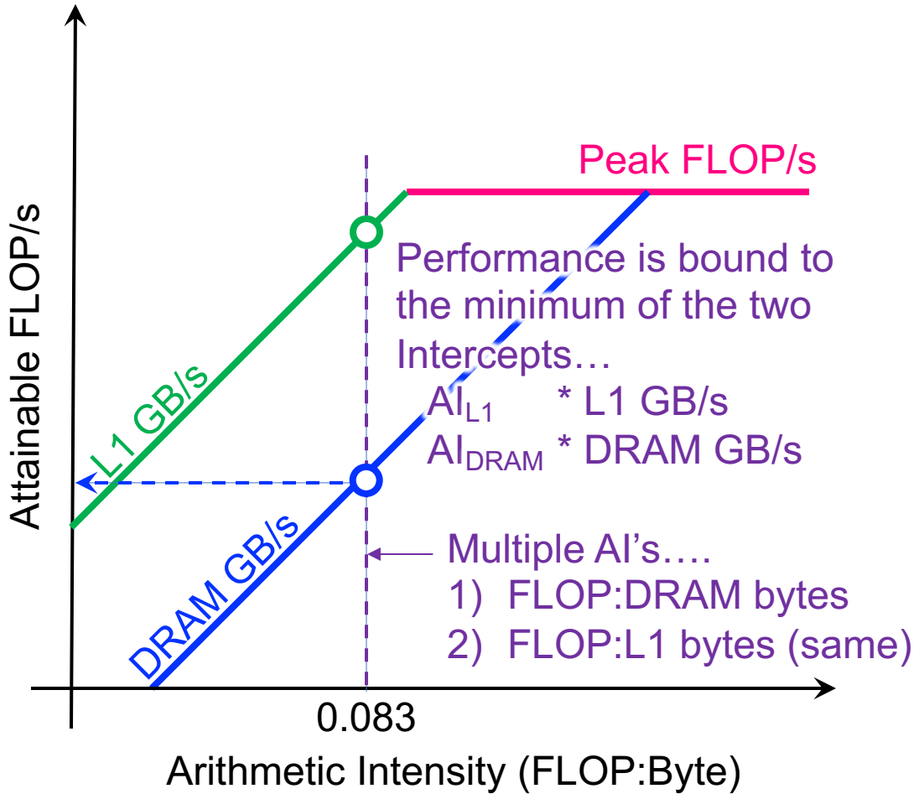
Example: STREAM

- 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

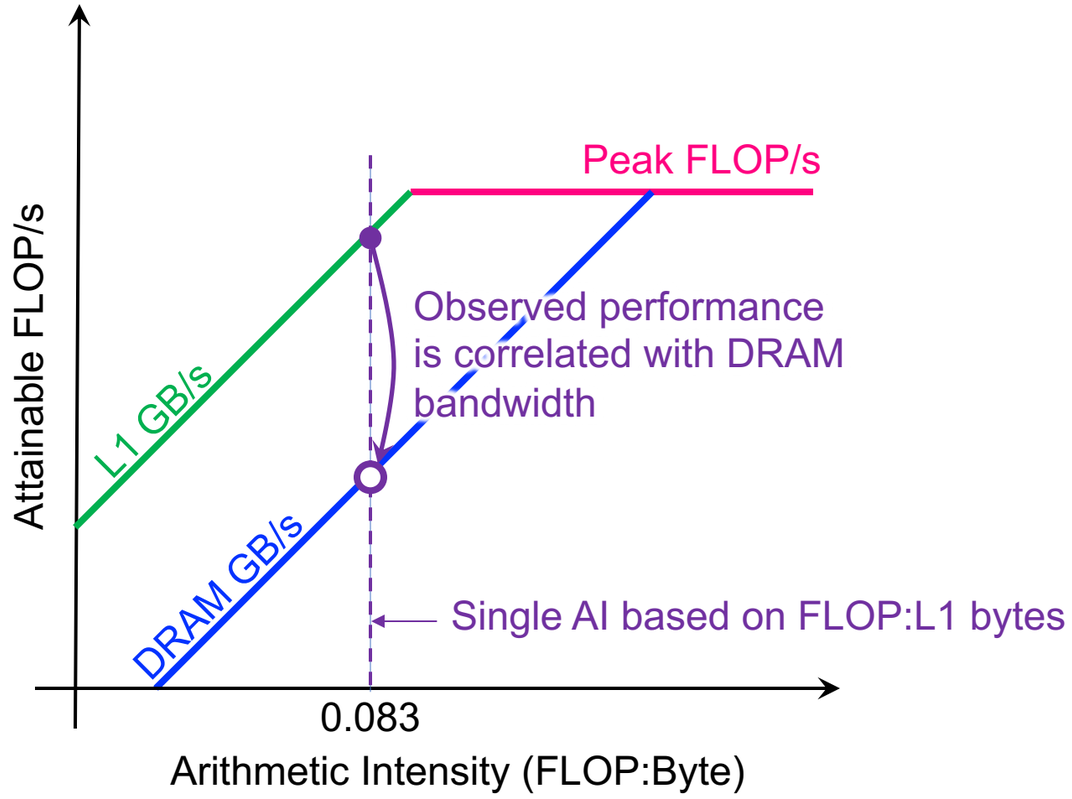
```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



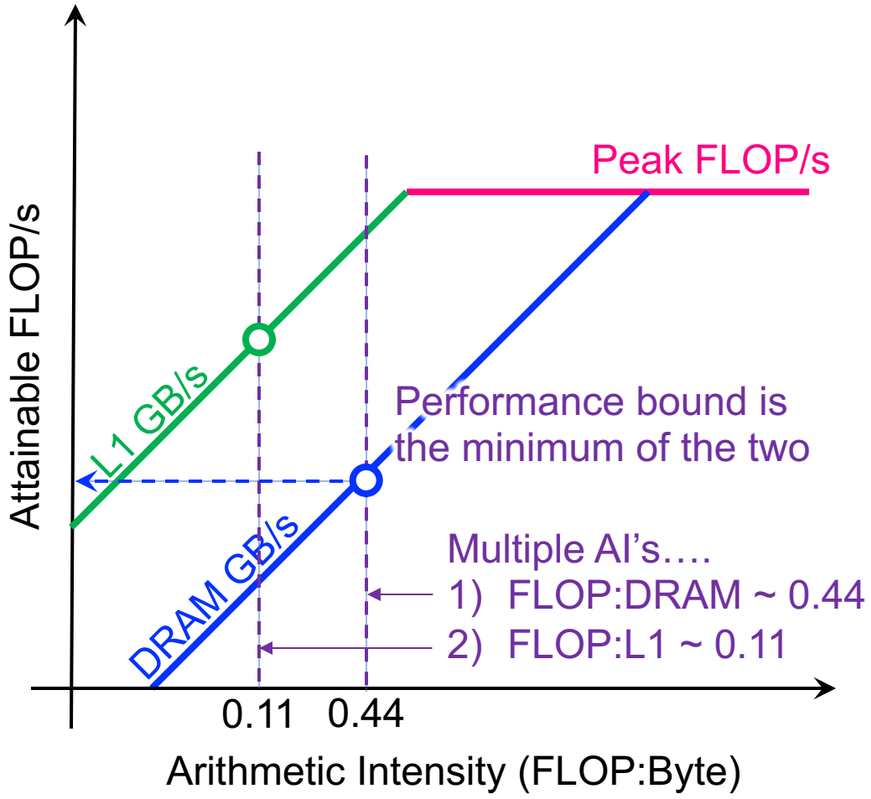
Example: 7-point Stencil (Small Problem)

- 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

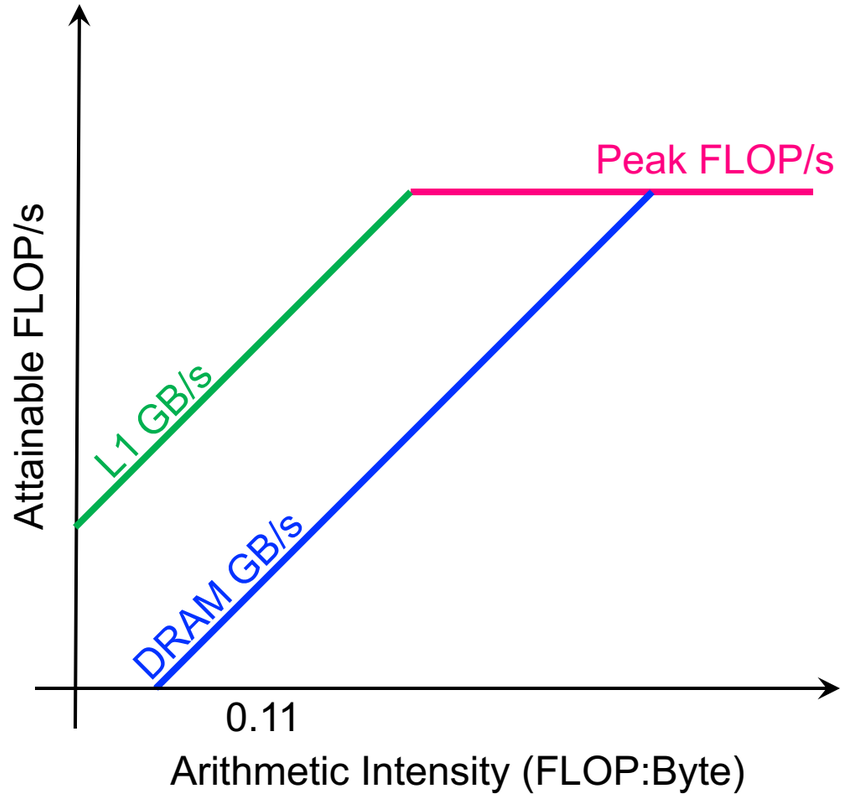
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk          ]
                 + old[ijk-1       ]
                 + old[ijk+1       ]
                 + old[ijk-jStride]
                 + old[ijk+jStride]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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

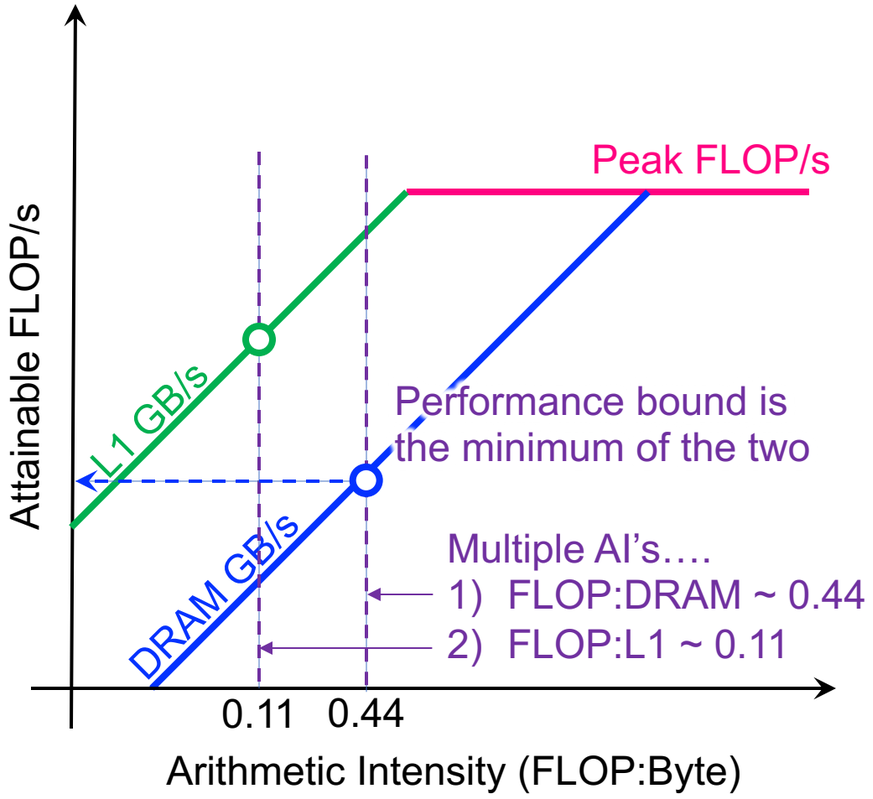


Cache-Aware Roofline

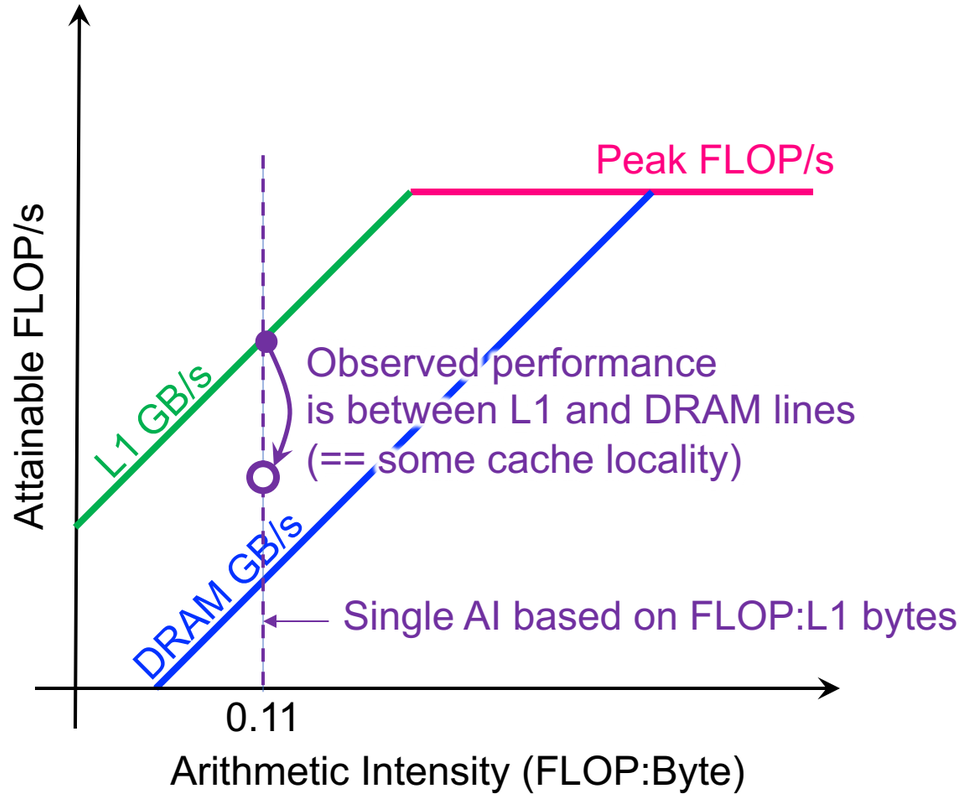


Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

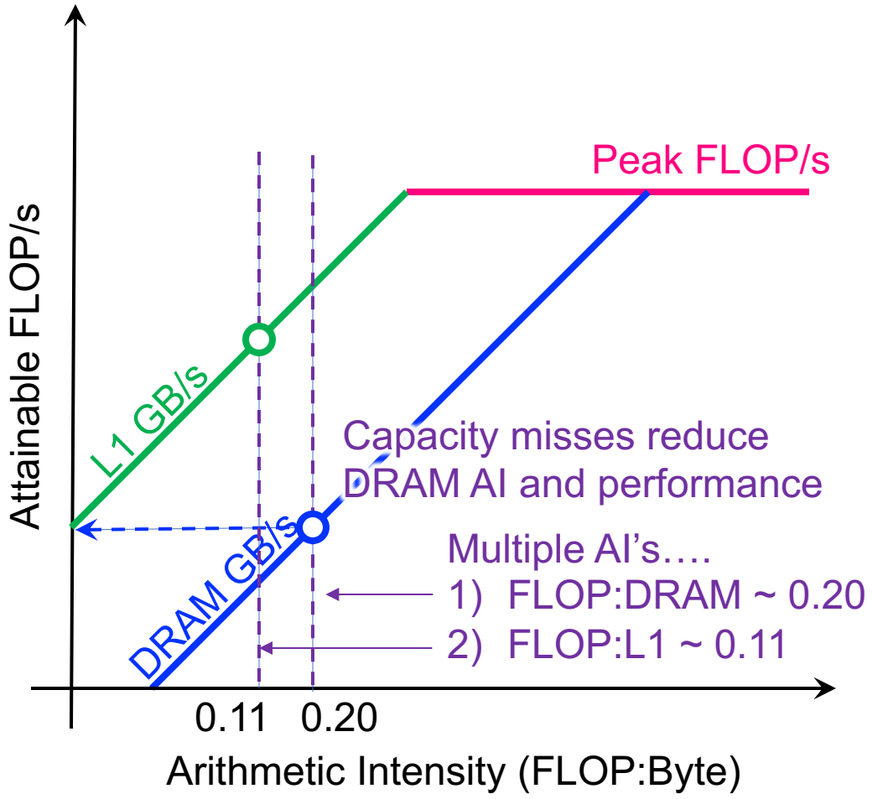


Cache-Aware Roofline

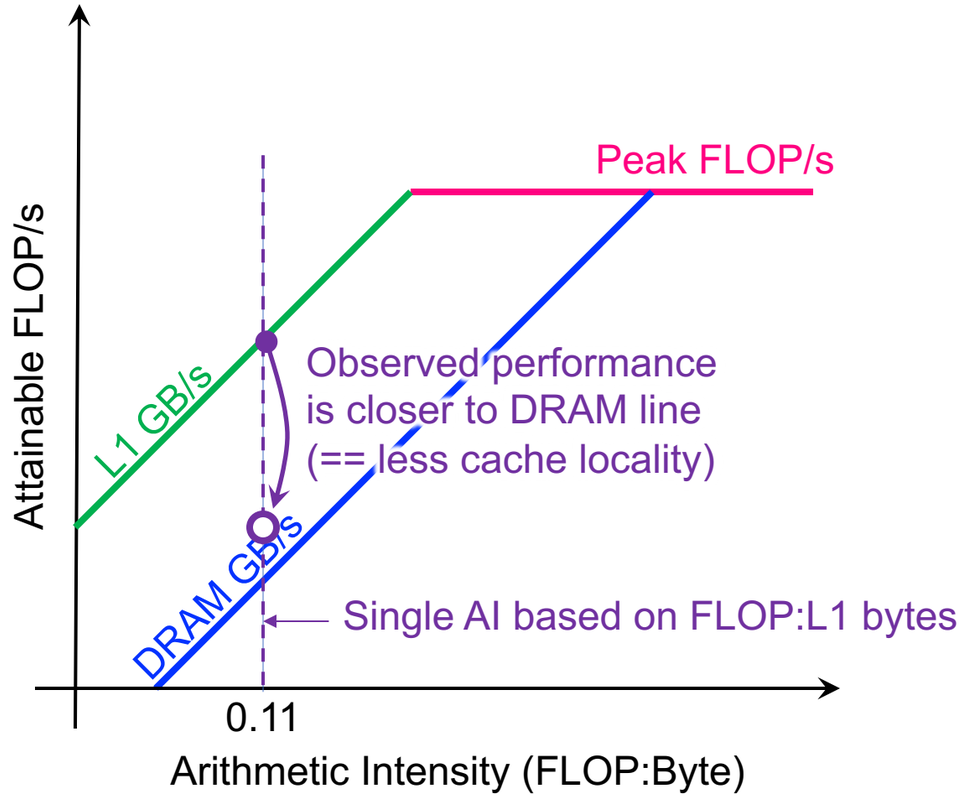


Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

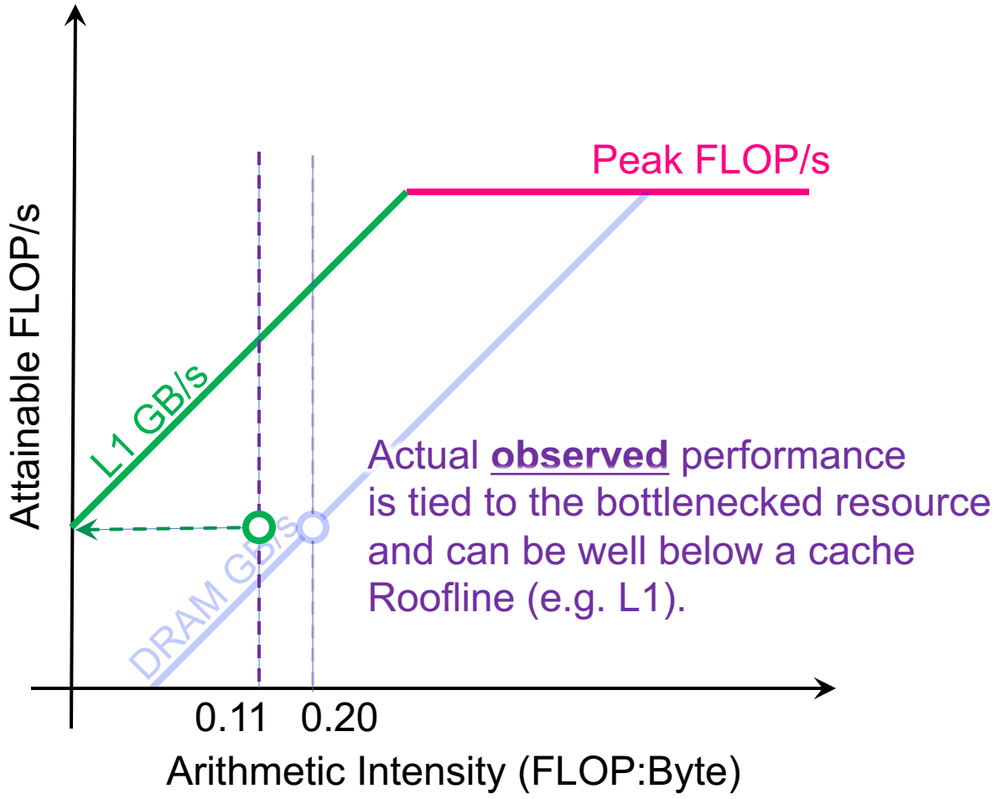


Cache-Aware Roofline

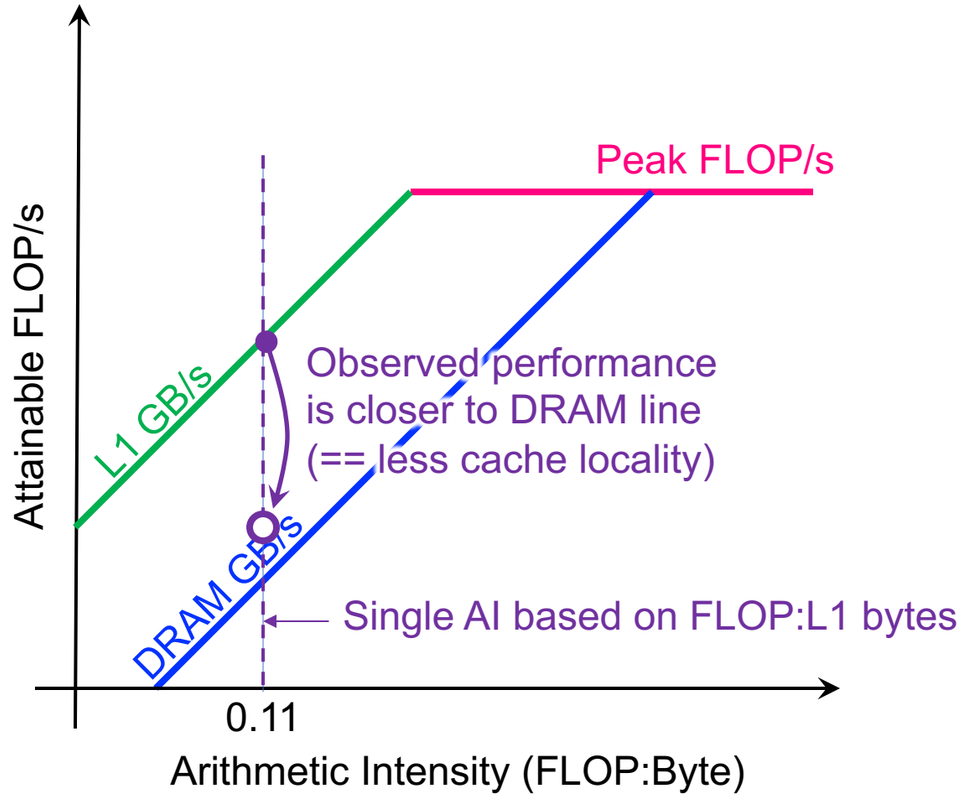


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

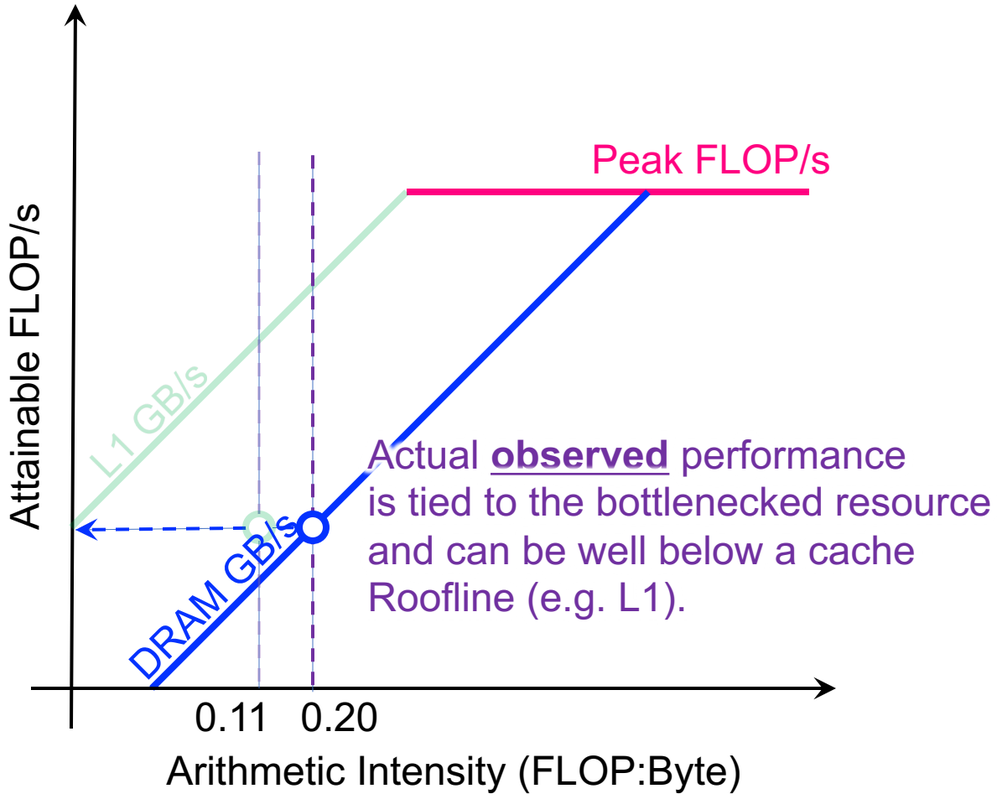


Cache-Aware Roofline



Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Cache-Aware Roofline