BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Performance Modeling and Analysis

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

Acknowledgements

Jack Deslippe, Charlene Yang, Doug Doerfler, Matt Cordery, Khaled Ibrahim, Lenny Oliker, Protonu Basu, Terry Ligocki, Brian Van Straalen (LBL), Linda Lo (formerly Utah), Zakhar Matveev (Intel), Roman Belenov (Intel)

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.

Introduction to Performance Modeling

Why Use Performance Models or Tools?

- Understand performance 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 software optimizations
- **Determine when we're done optimizing**
  - Assess performance relative to machine capabilities
  - Motivate need for algorithmic changes

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

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

DAX
N

```
#pragma omp parallel for
for(i=0;i<N;i++){
for(j=0;j<N;j++){
 double Cij=0;
  r(k=0;k<N;k++){
  ij += A[i][k] * B[k][j];

  ][j] = sum;
```

GEMM: O(N^3) complexity where N is the number of rows (equatic

What are the scaling constants?

FFTs: O(NlogN) in the number of

CG: O($N^{1.33}$) in the number of

MG: O(N) in the number of ele

N-body: O(N^2) in the number o

Why did we depart from ideal scaling?

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		

Which is more expensive…

Performing FLOPs, or

Moving words from memory

[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…

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

- …and arithmetic intensity as the ratio of…

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

Operation	FLOPs	Data	AI (ideal)
DAXPY	O(N)	O(N)	O(1)
DGEMV	$O(N^2)$	$O(N^2$	O(1)
DGEMM	$O(\ ^3)$		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

!

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)

Computational Depth

- Parallel machines incur substantial overheads on synchronization (shared memory), point-to-point communication, reductions, and broadcasts.
- We can classify algorithms by **depth** (max depth of the algorithm's dependency chain)

➢ **If dependency chain crosses process boundaries, we incur substantial overheads.**

Operation	FLOPs	Data	AI [illegible]eal)	Depth
DAXPY	O(N)	O([illegible])	[illegible]1)	O(1)
DGEMV	O(N^2)	O[illegible]	[illegible]1)	O(logN)
DGEMM	O([illegible]	[illegible]	[illegible]N)	O(logN)
FFTs	O(Nlo[illegible]	[illegible]	[illegible]	O(logN)
CG	O(N^1[illegible]	[illegible]	[illegible]	[illegible]logN)
MG	[illegible]	[illegible]	[illegible]	O(logN)
N-body	[illegible]	[illegible]	[illegible]	O(logN)

Overheads can dominate at high concurrency or small problems

!

BERKELEY LAB

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…

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.

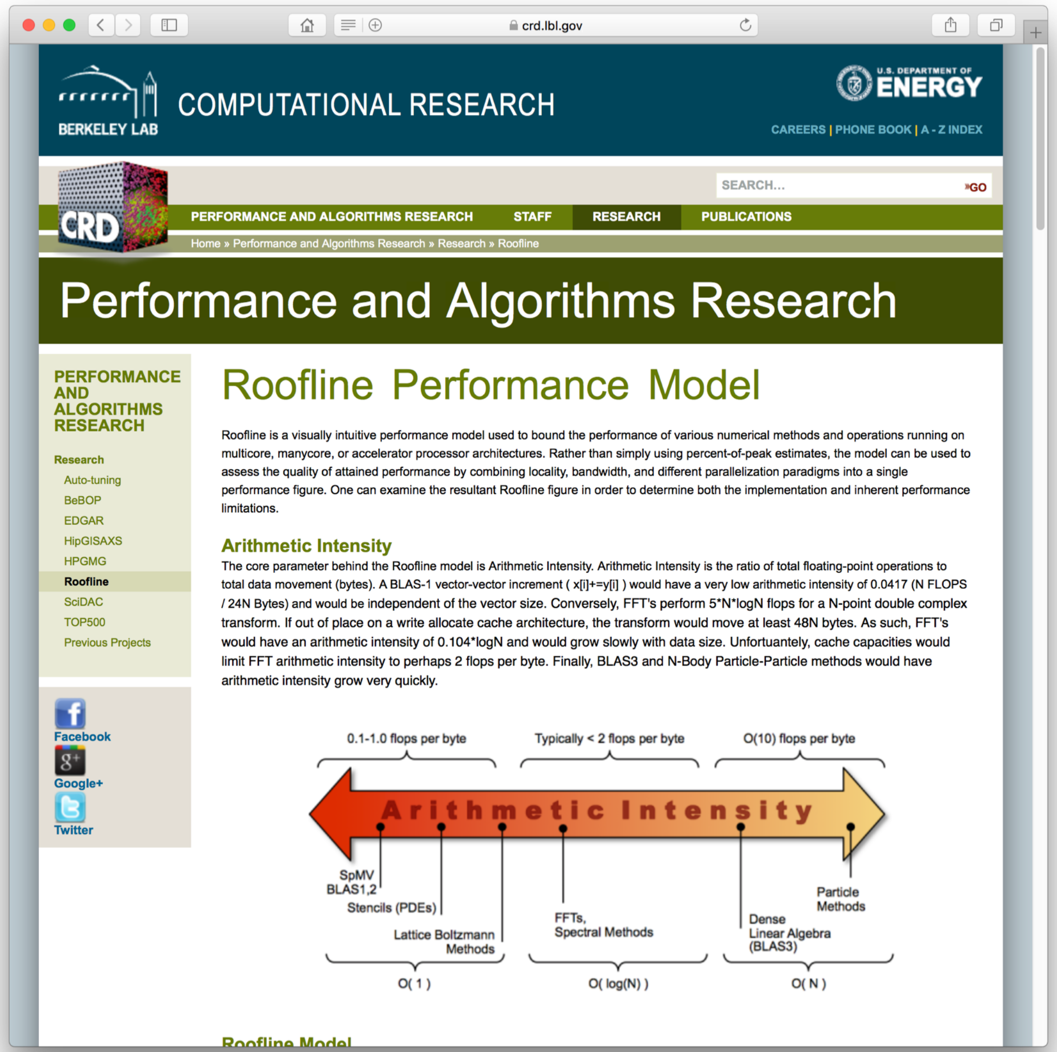
Implications of Architectural Evolution…

- 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**

Roofline Model

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



crd.lbl.gov

BERKELEY LAB COMPUTATIONAL RESEARCH

U.S. DEPARTMENT OF ENERGY

CAREERS | PHONE BOOK | A - Z INDEX

SEARCH... »GO

CRD

PERFORMANCE AND ALGORITHMS RESEARCH   STAFF   RESEARCH   PUBLICATIONS

Home » Performance and Algorithms Research » Research » Roofline

Performance and Algorithms Research

PERFORMANCE AND ALGORITHMS RESEARCH

Research
- Auto-tuning
- BeBOP
- EDGAR
- HipGISAXS
- HPGMG
- Roofline
- SciDAC
- TOP500
- Previous Projects

Facebook
Google+
Twitter

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.

BERKELEY LAB

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 band
Depth	OMP O
MPI Message Size	Network B
MPI Send:Wait ratio	Network Ga
#MPI Wait's	Network Latency

Rooflin

Use the right model !

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

BERKELEY LAB

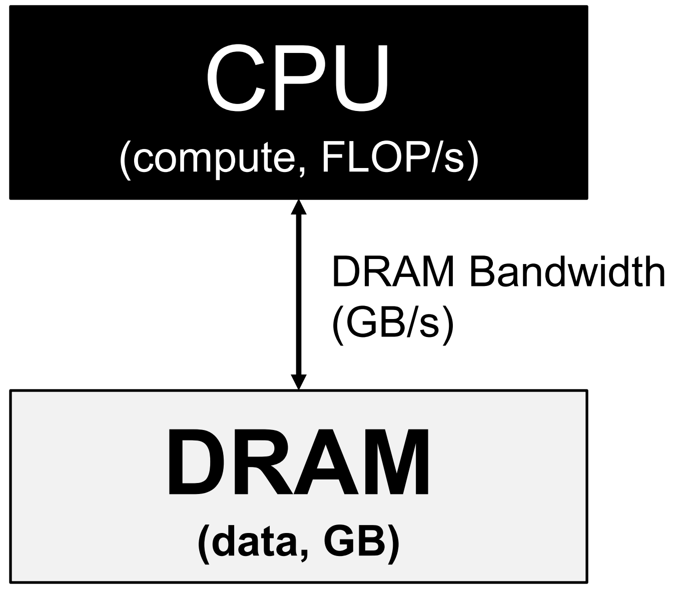
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Introduction to the Roofline Model

(DRAM) Roofline

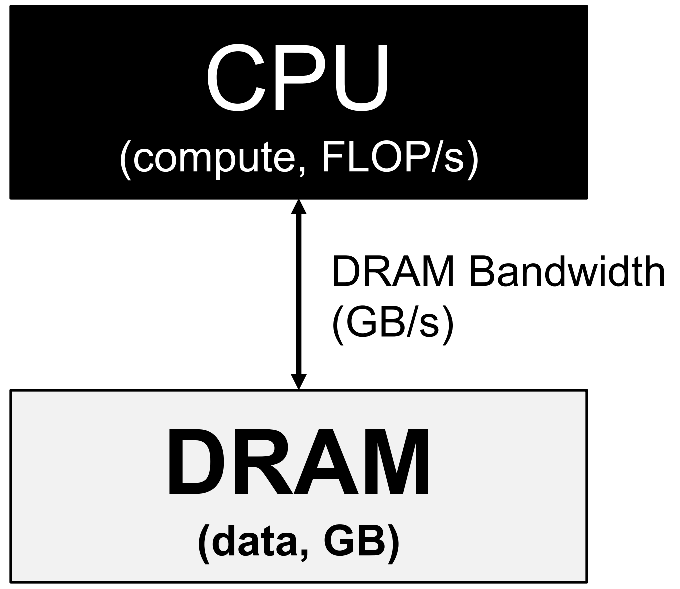
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

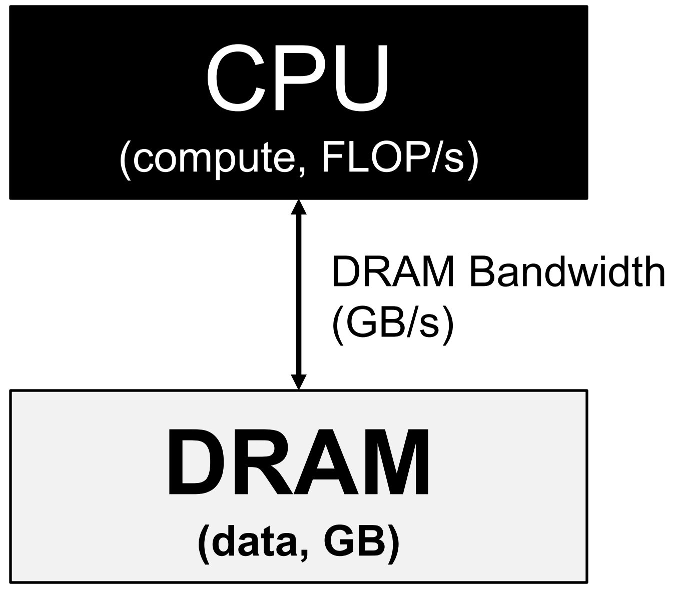
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

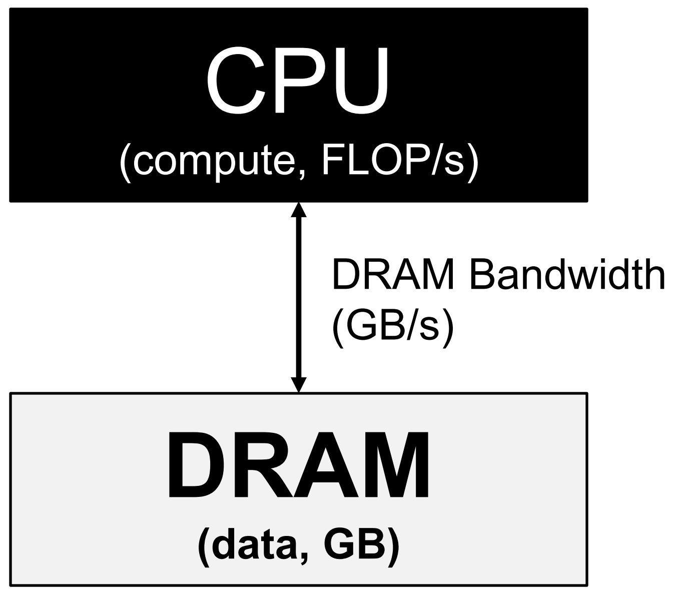
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(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)



$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases}$$

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

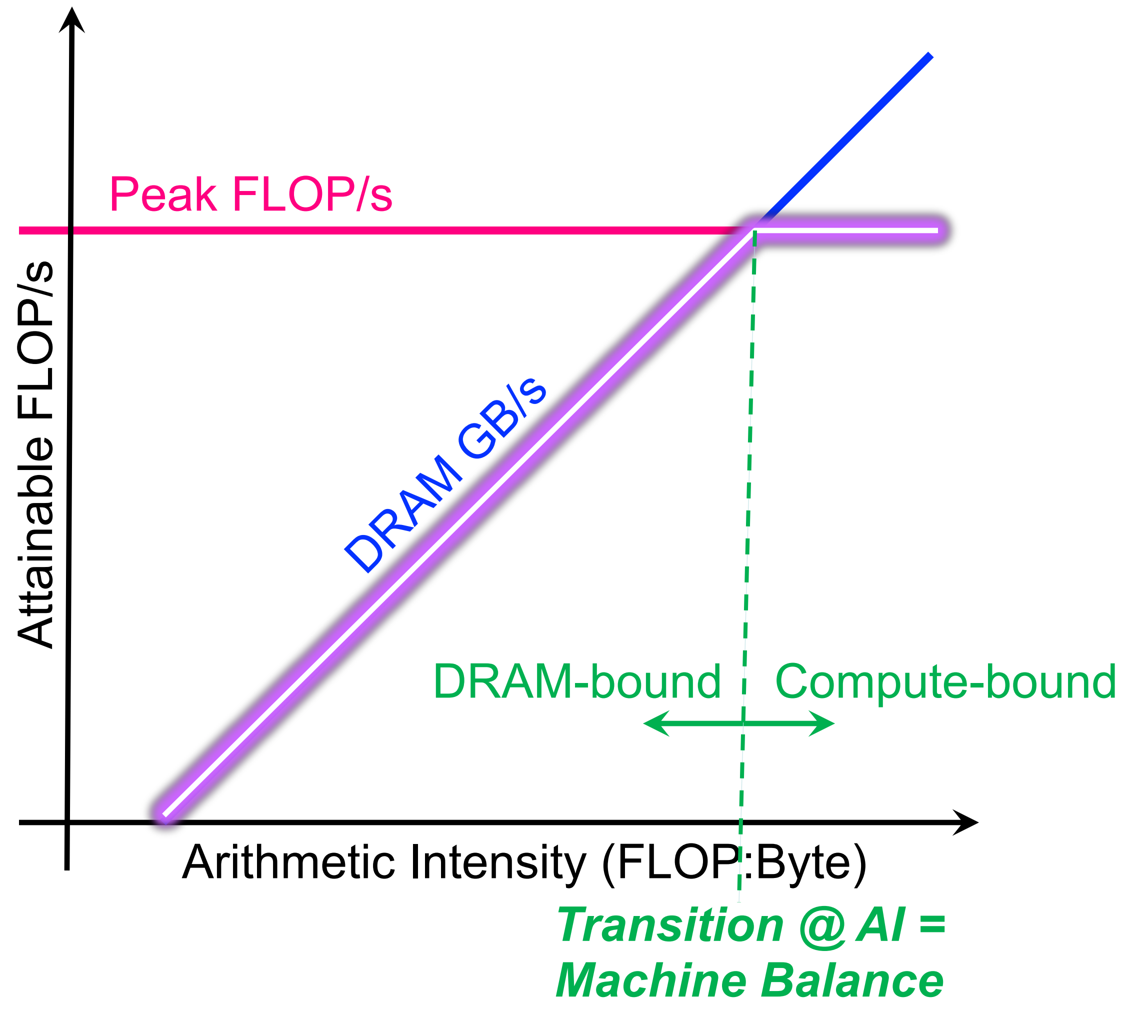
Arithmetic Intensity

- 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…)


Peak FLOP/s
Attainable FLOP/s
DRAM GB/s
DRAM-bound
Compute-bound
Arithmetic Intensity (FLOP:Byte)
Transition @ AI = Machine Balance

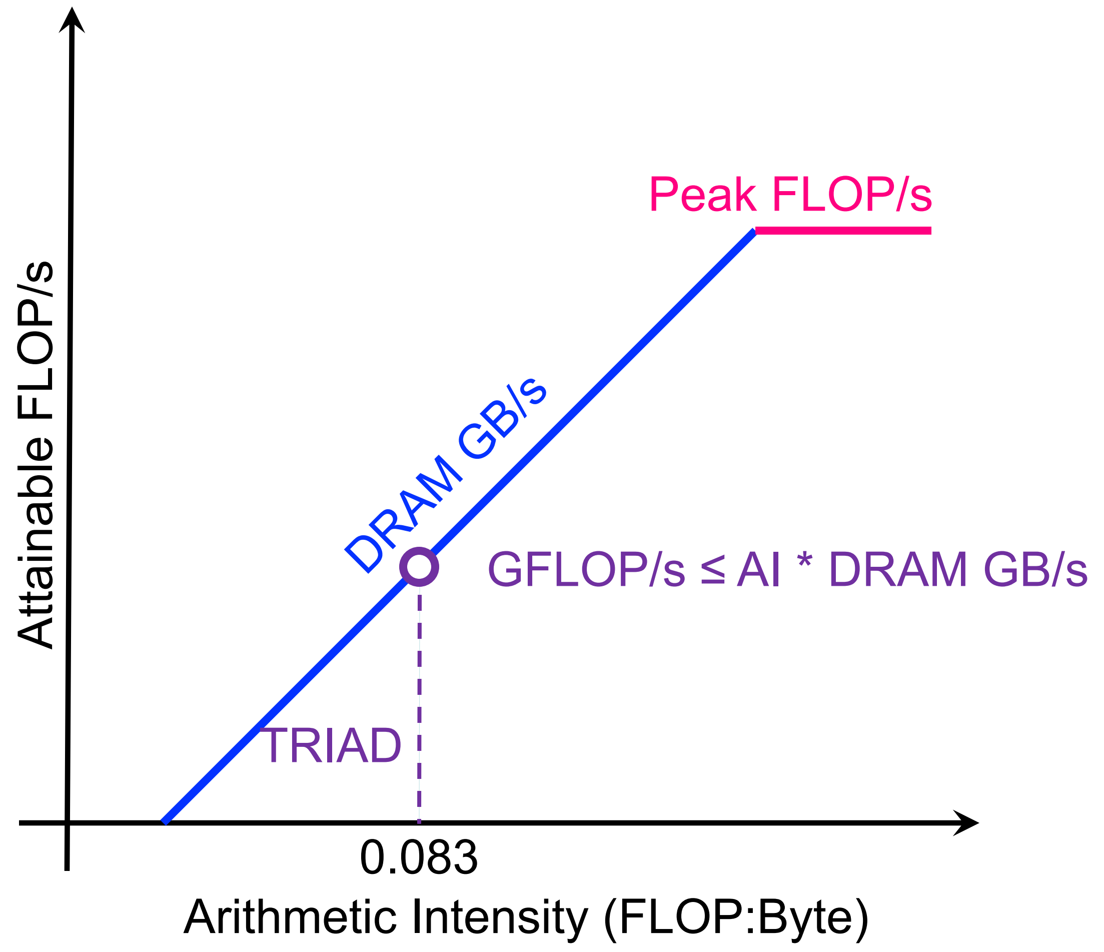
Roofline Example #1

- 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];
}
```

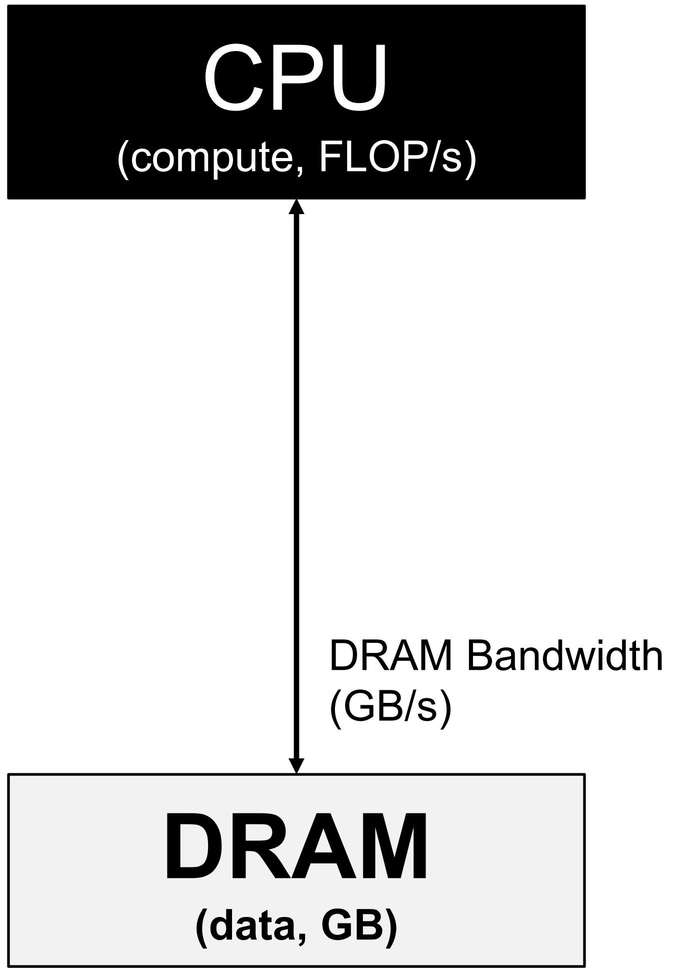
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 flops per byte == Memory bound**



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)**

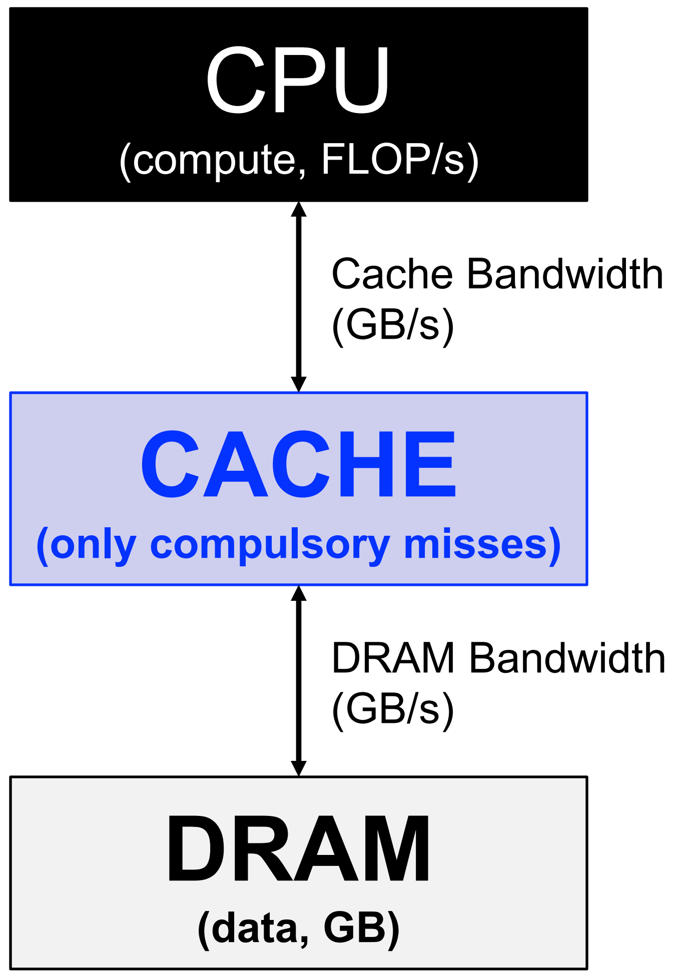
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                    + old[k  ][j  ][i-1]
                    + old[k  ][j  ][i+1]
                    + old[k  ][j-1][i  ]
                    + old[k  ][j+1][i  ]
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

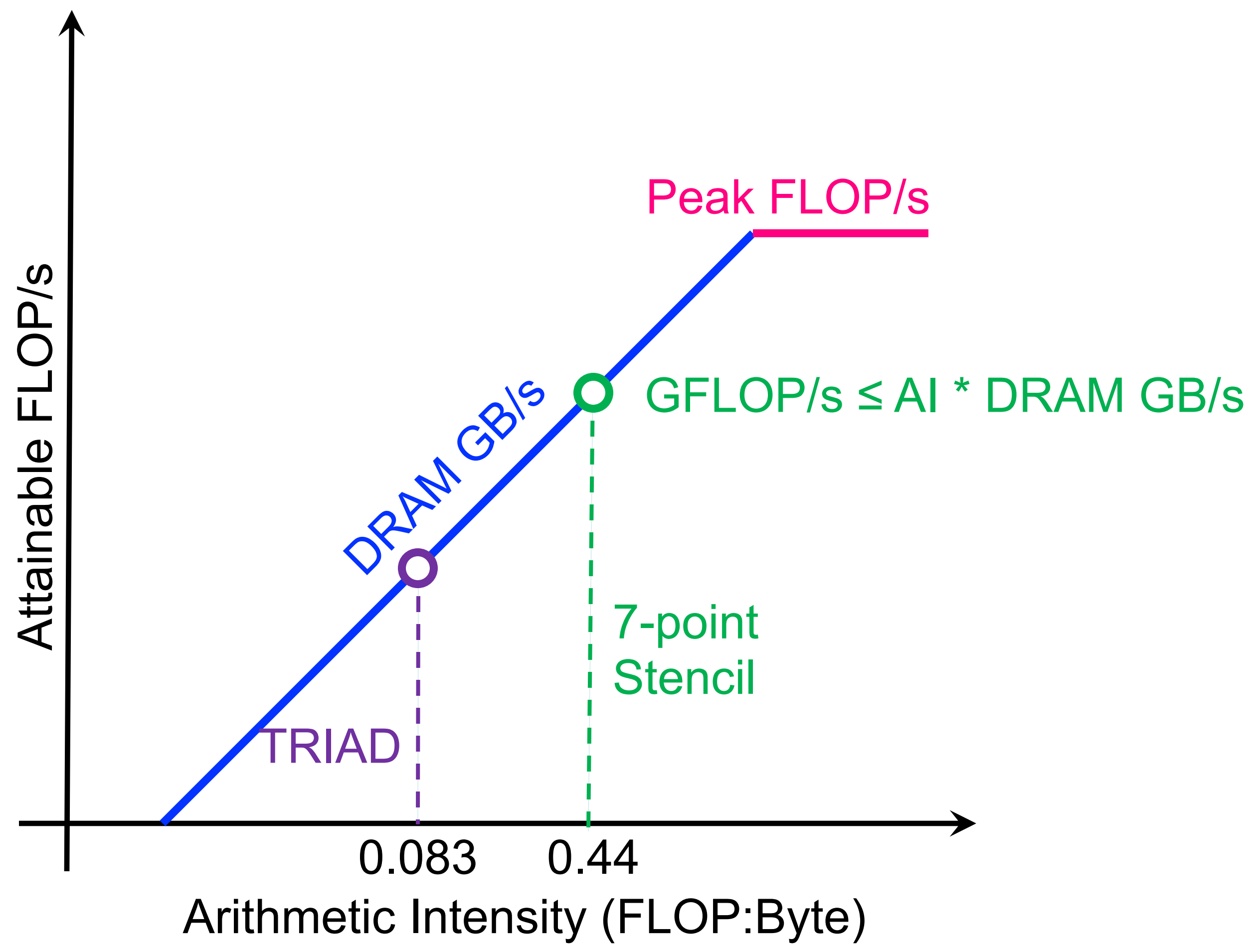
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                    + old[k  ][j  ][i-1]
                    + old[k  ][j  ][i+1]
                    + old[k  ][j-1][i  ]
                    + old[k  ][j+1][i  ]
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



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

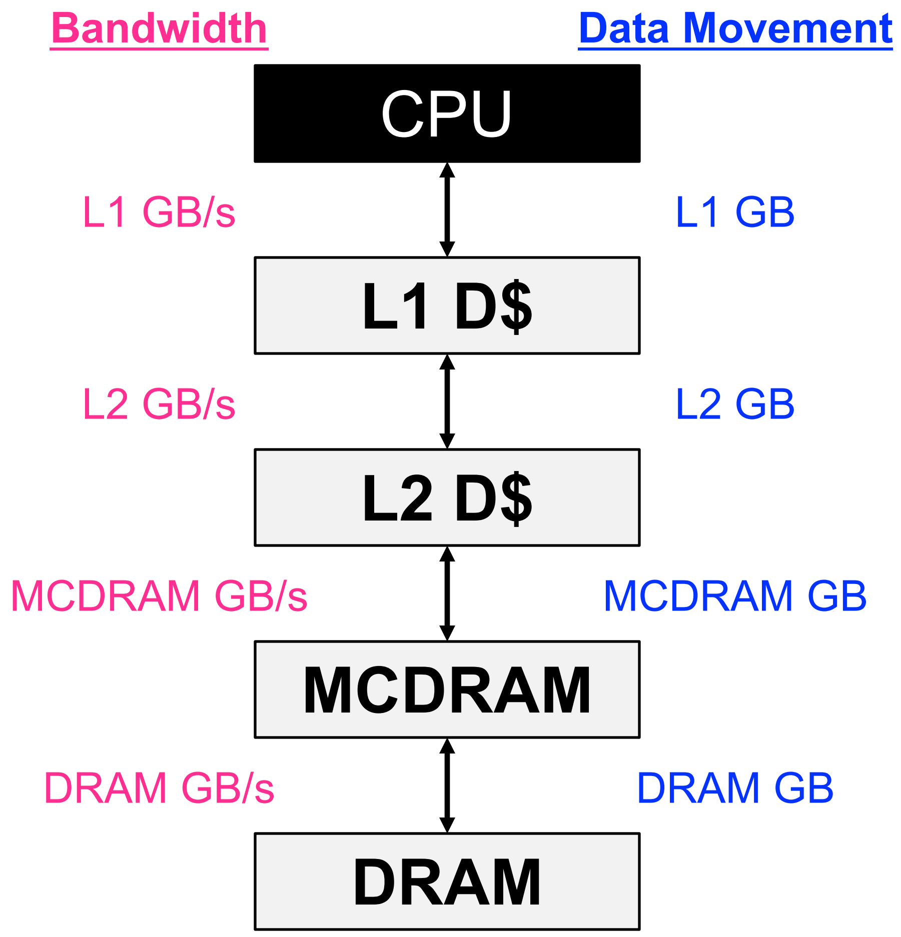
- **No, there can be other performance bottlenecks…**
  - Cache bandwidth / locality
  - Lack of vectorization / SIMDization
  - Load imbalance
  - …

Extending the Roofline:

Memory Hierarchy

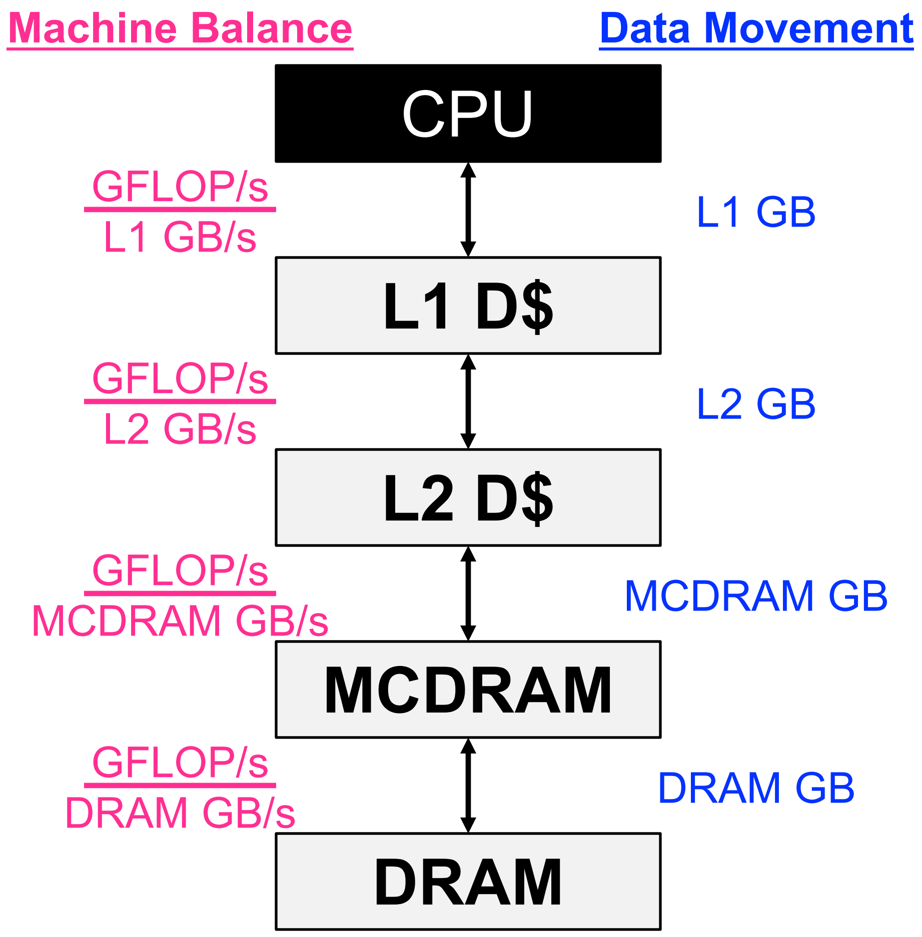
Hierarchical Roofline

- 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



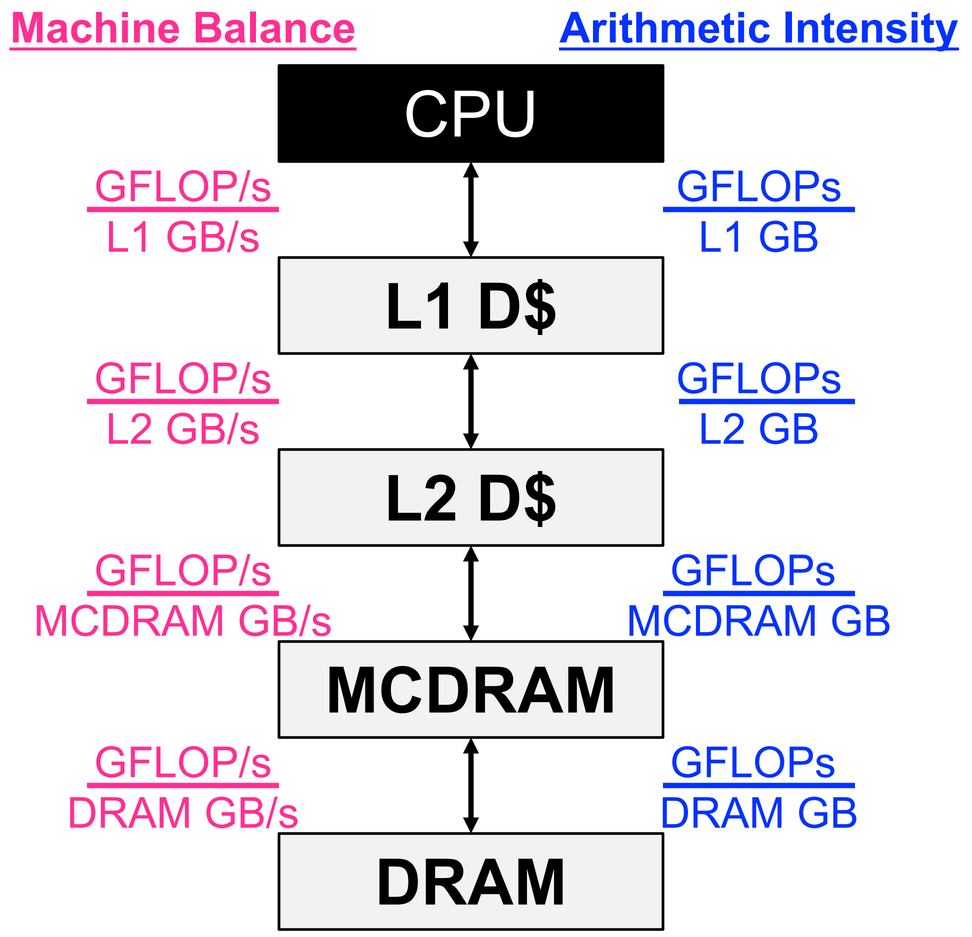
Hierarchical Roofline

- 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



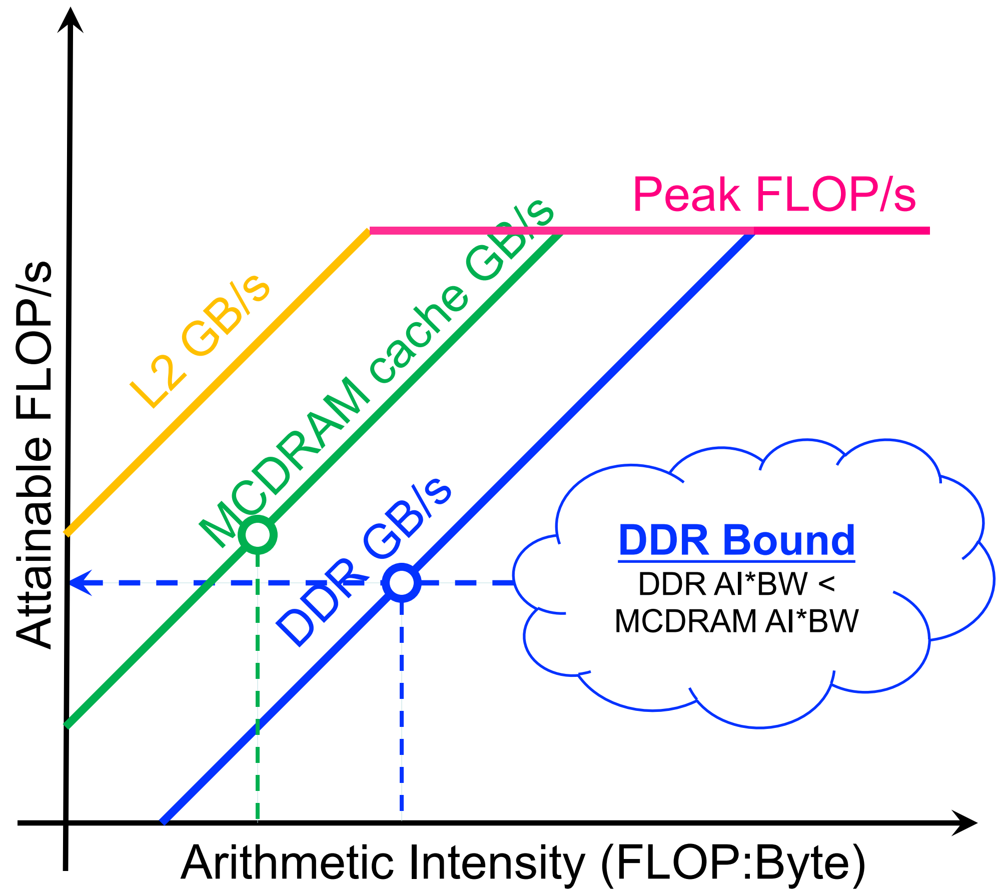
Hierarchical Roofline

- 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



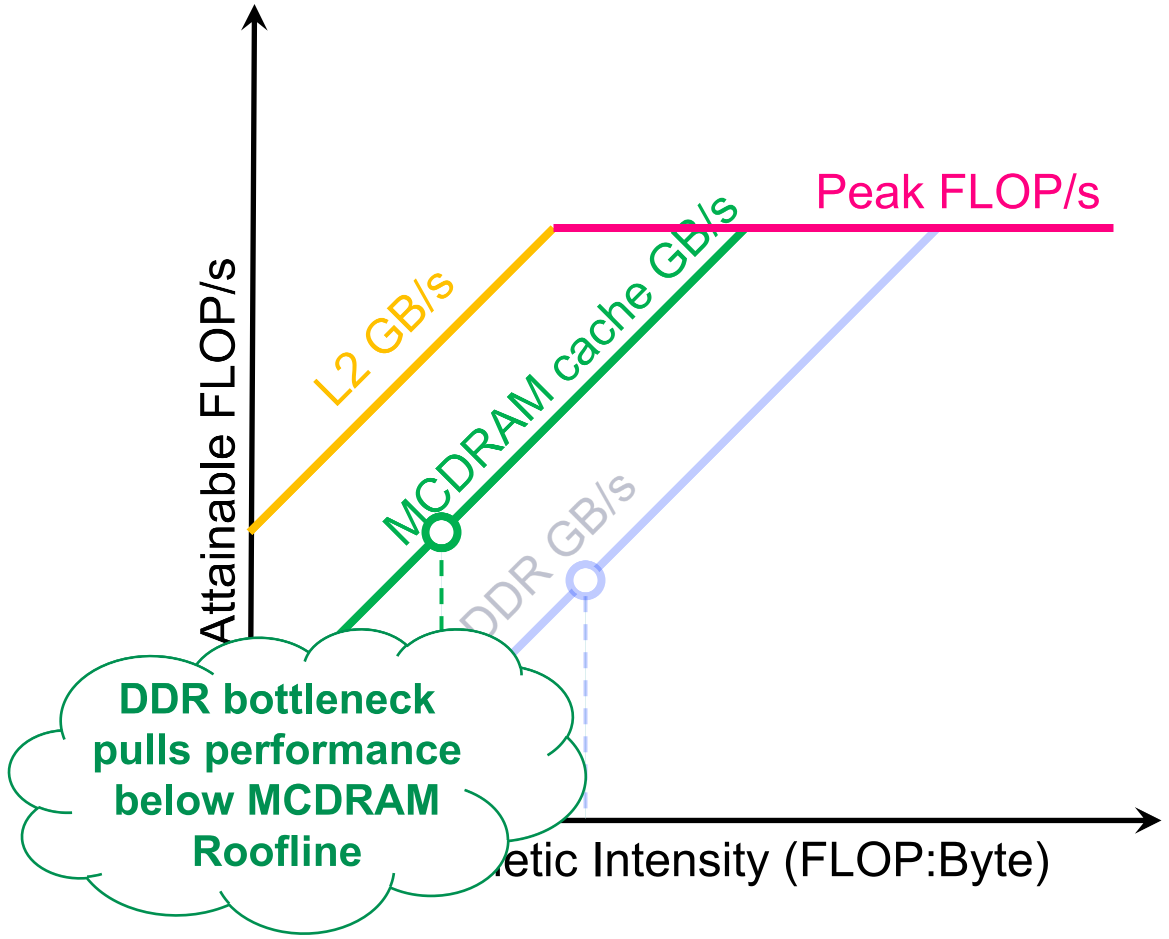
Hierarchical Roofline

- 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**



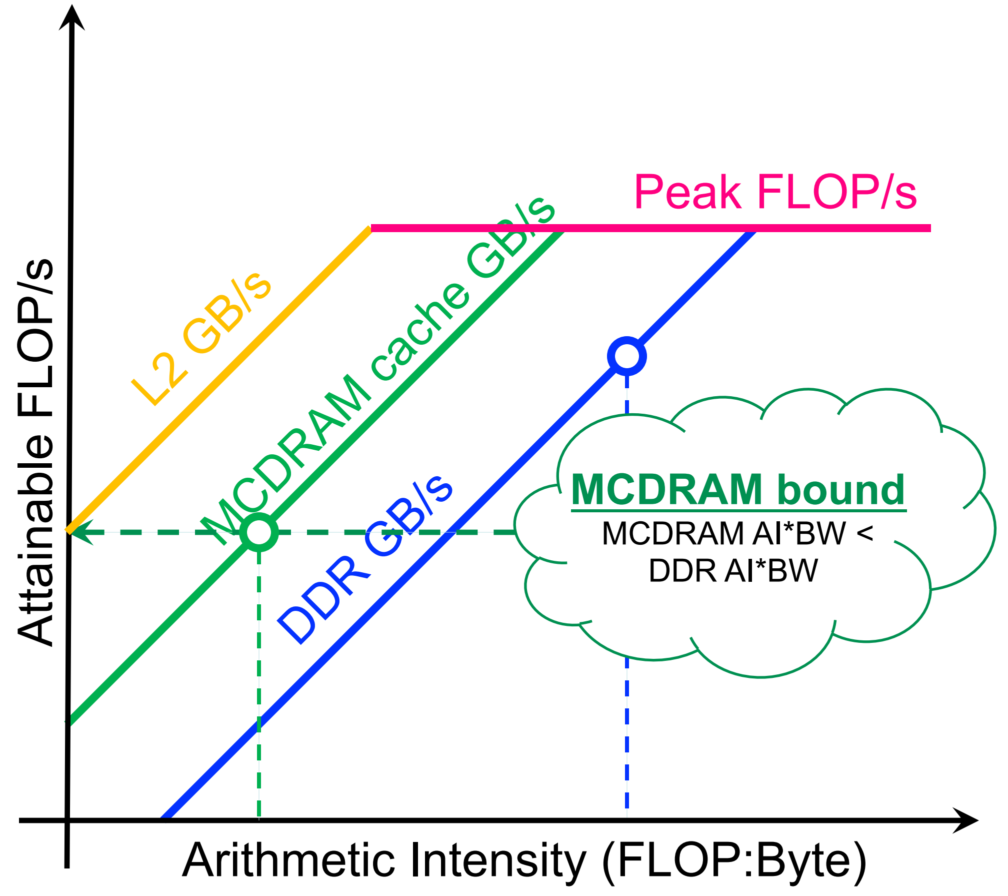
Hierarchical Roofline

- 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**


Peak FLOP/s
L2 GB/s
MCDRAM cache GB/s
DDR GB/s
Attainable FLOP/s
…etic Intensity (FLOP:Byte)
DDR bottleneck pulls performance below MCDRAM Roofline

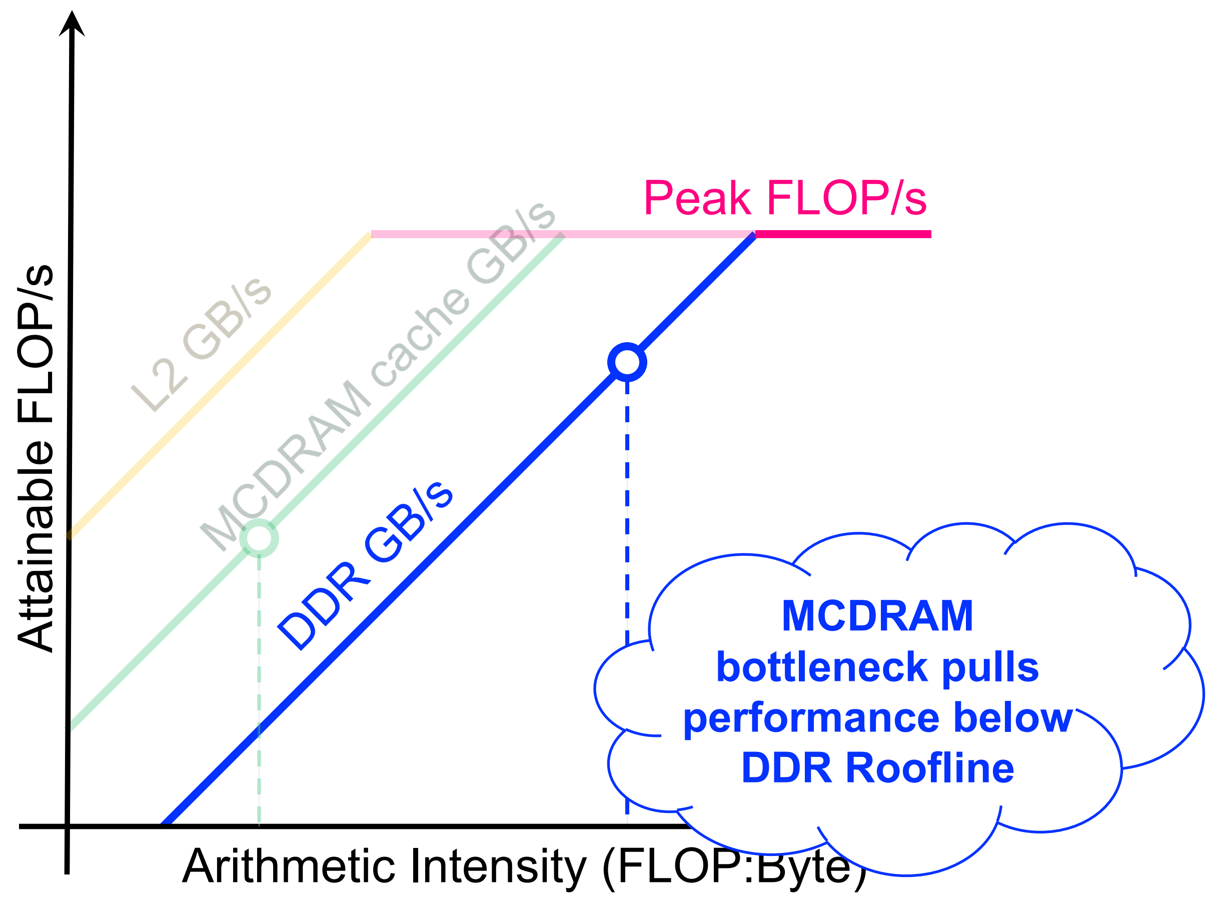
Hierarchical Roofline

- 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

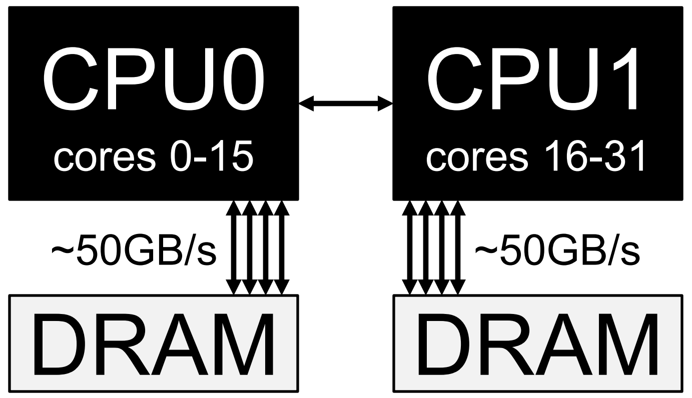
- Construct superposition of Rooflines…
  - Measure bandwidth
  - Measure AI for each level of memory
  - Although an loop nest may have multiple AI's and multiple bounds (flops, L1, L2, … DRAM)…
  - **… performance is bound by the minimum**

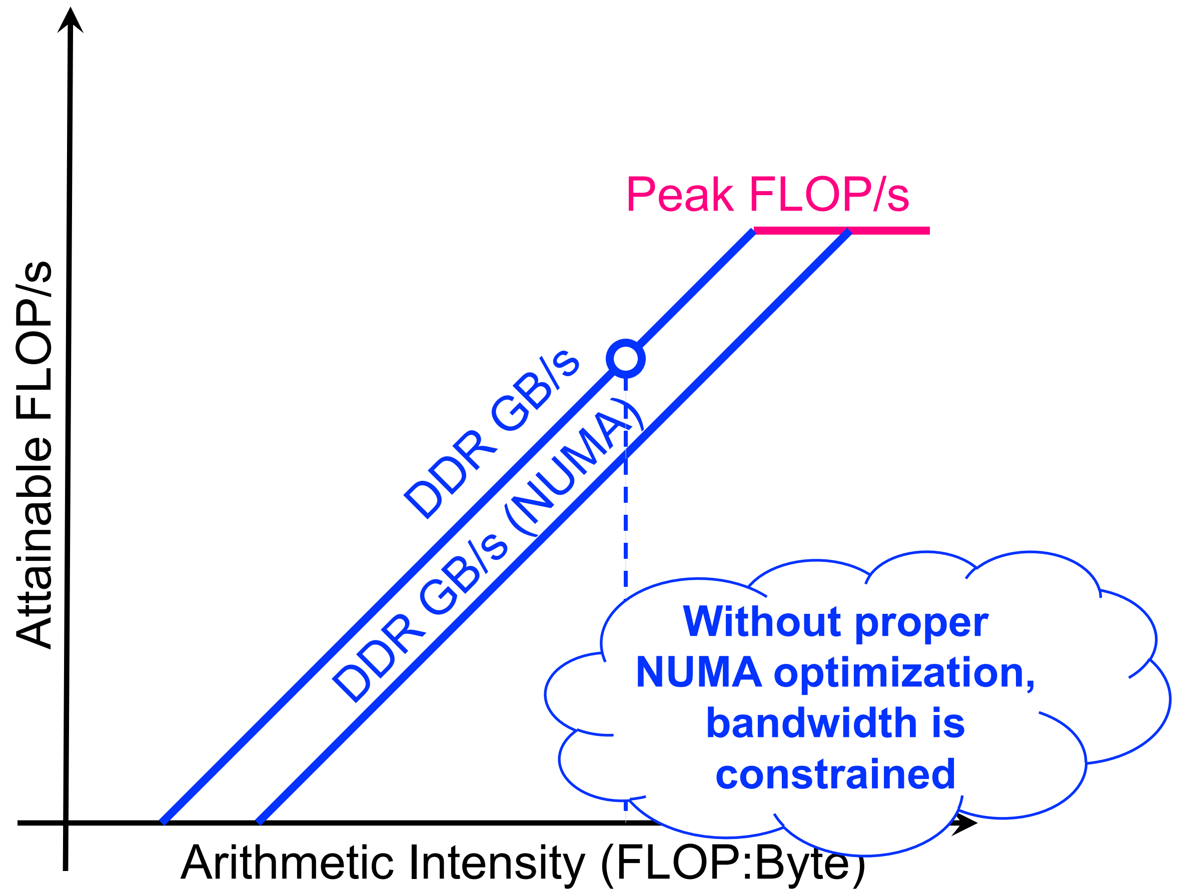


Peak FLOP/s
L2 GB/s
MCDRAM cache GB/s
DDR GB/s
Attainable FLOP/s
Arithmetic Intensity (FLOP:Byte)
MCDRAM bottleneck pulls performance below DDR Roofline

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

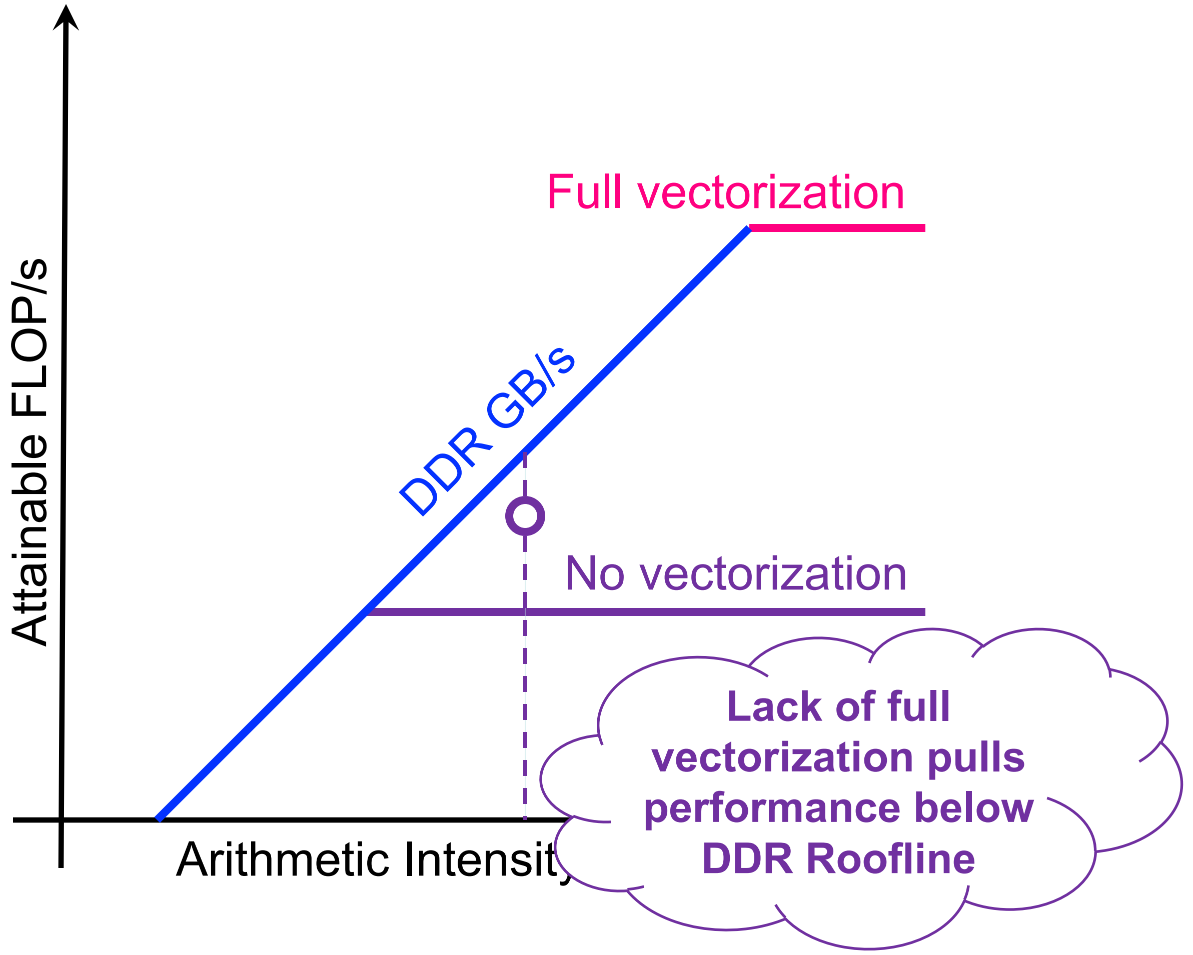
In-Core Parallelism

- 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

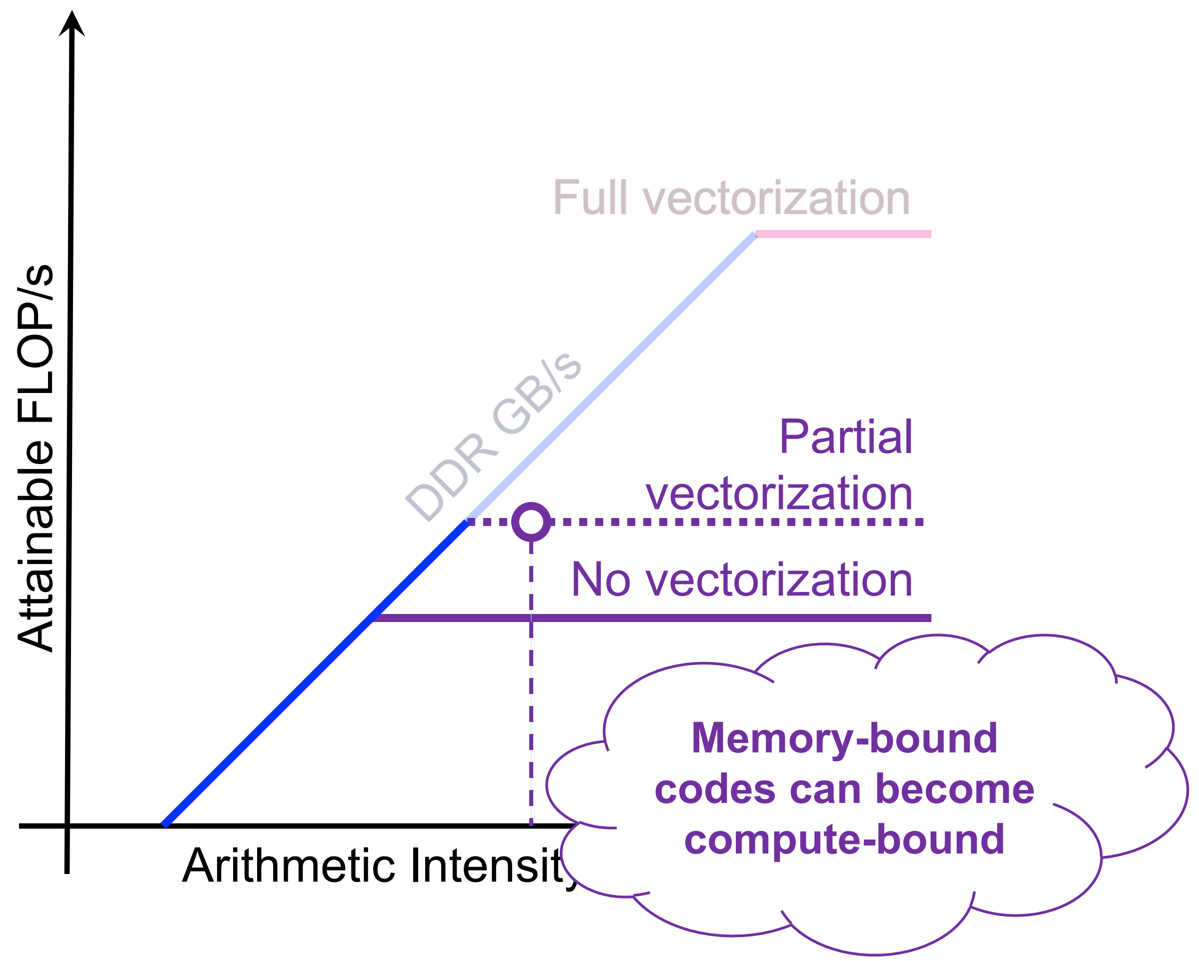
Data Parallelism (e.g. SIMD)

- 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**



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
  - ➢ **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*

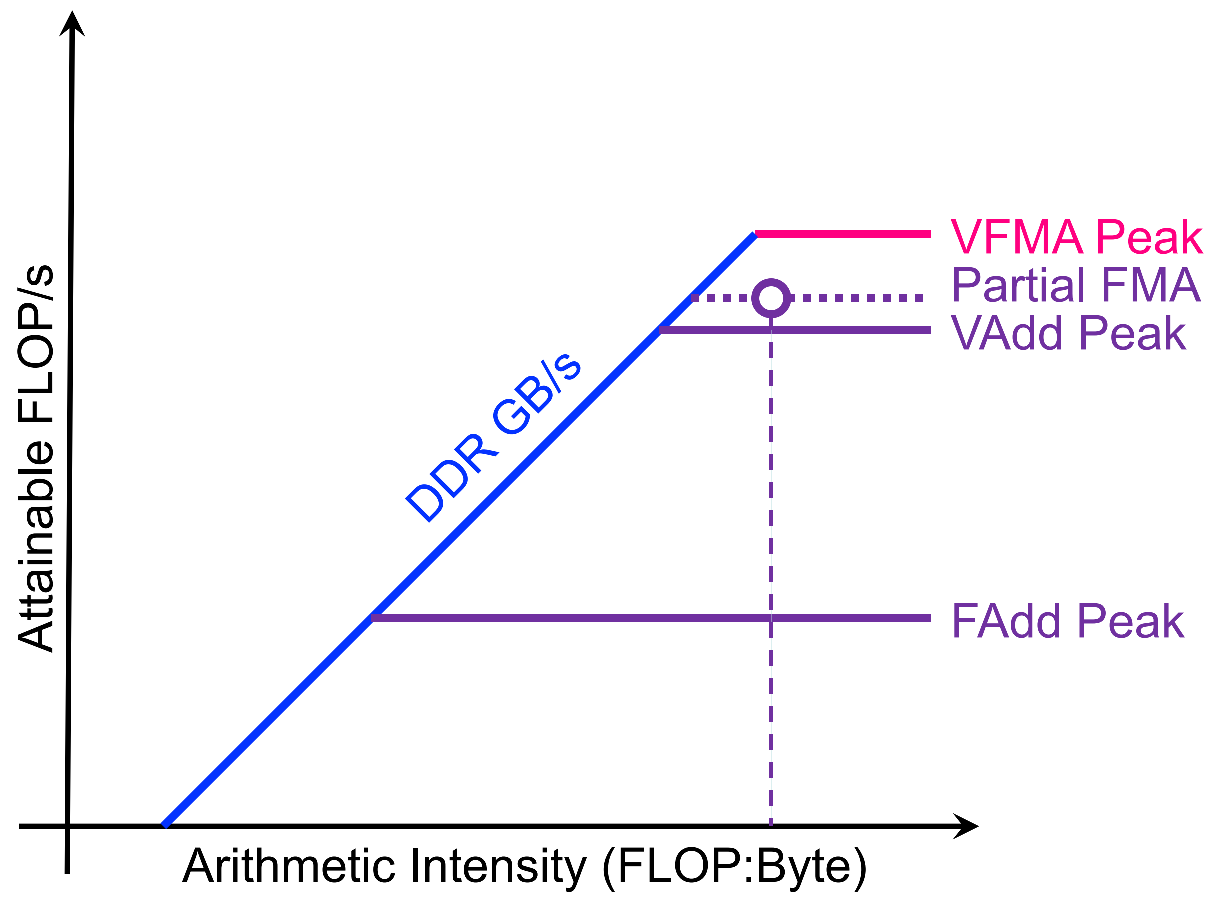
➢ **Performance is now a weighted average of scalar, vector, FMA, and WMMA operations.**

Return of CISC

- Total lack of FMA reduces performance by 2x on KNL.

  (4x on Haswell)

- In reality, applications are a mix of FMA, FAdd, and FMul.
  - Performance is a weighted average
  - ➢ **There is an implicit ceiling based on this weighted average**



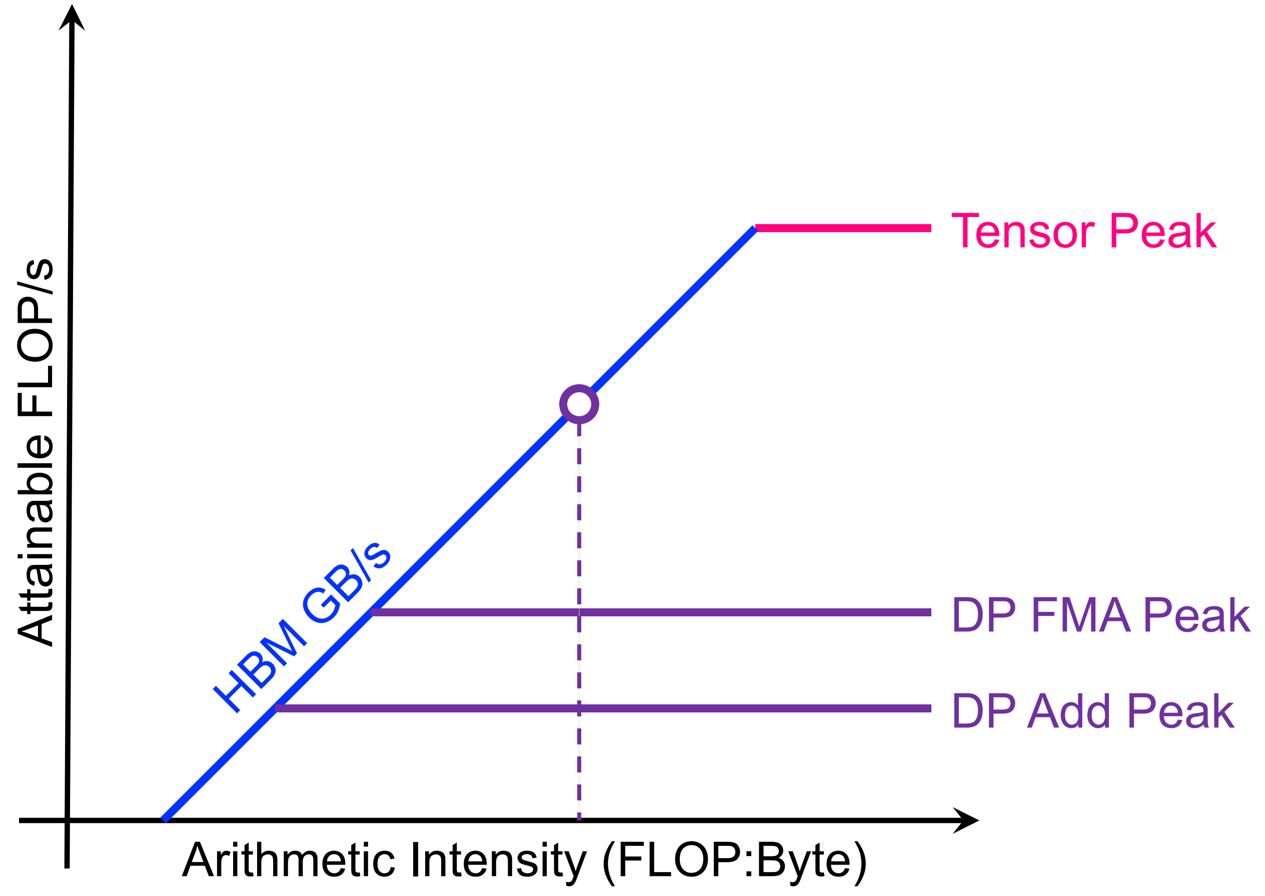
Return of CISC

- On Volta, Tensor cores can provide 100TFLOPs of FP16 performance

  (vs. 7.5 TFLOPS for DP FMA)
- Observe, machine balance has now grown to …

  100 TFLOP/s / 800 GB/s

  = 250 FP16 ops per word !!



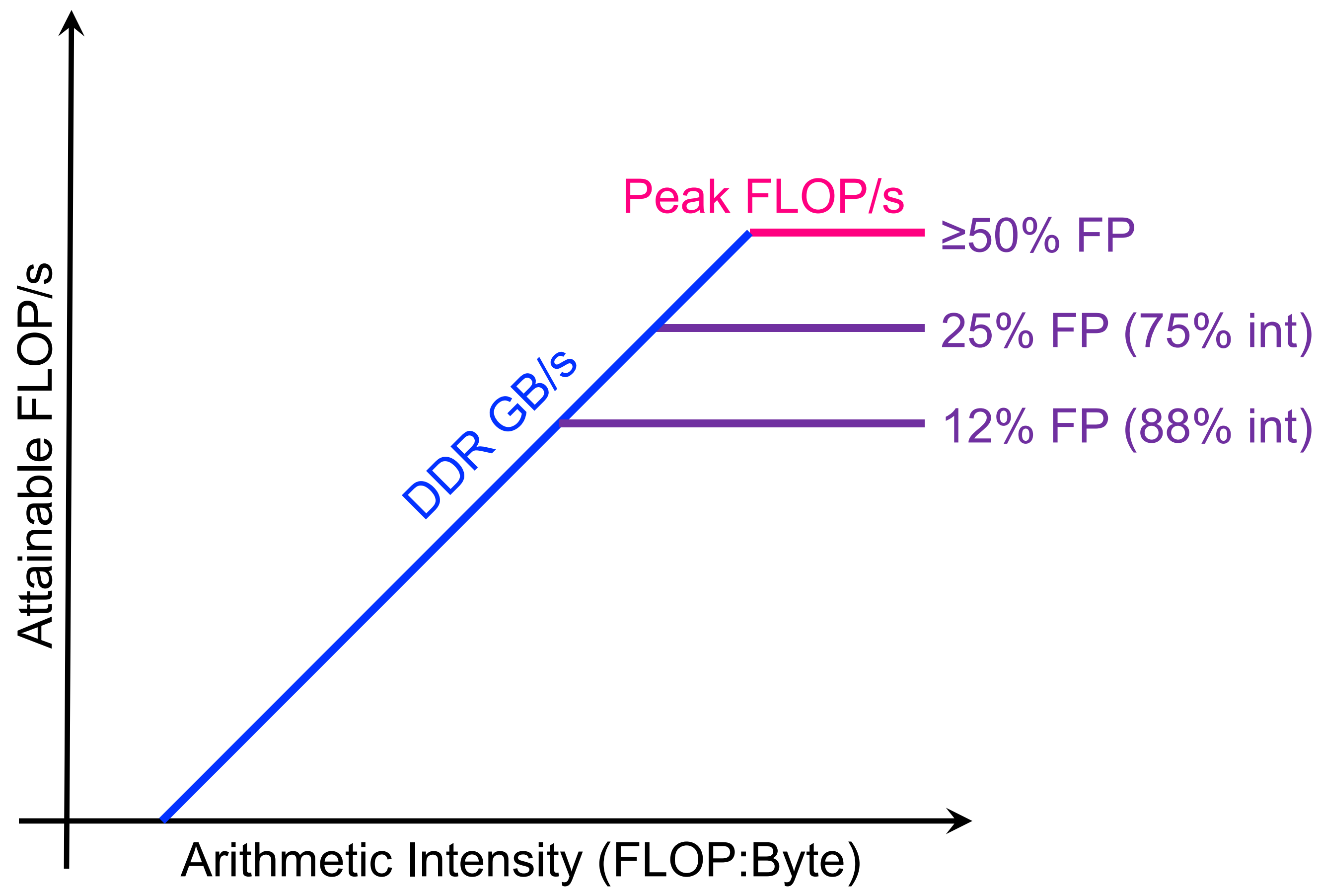
Superscalar vs. Instruction mix

- 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**

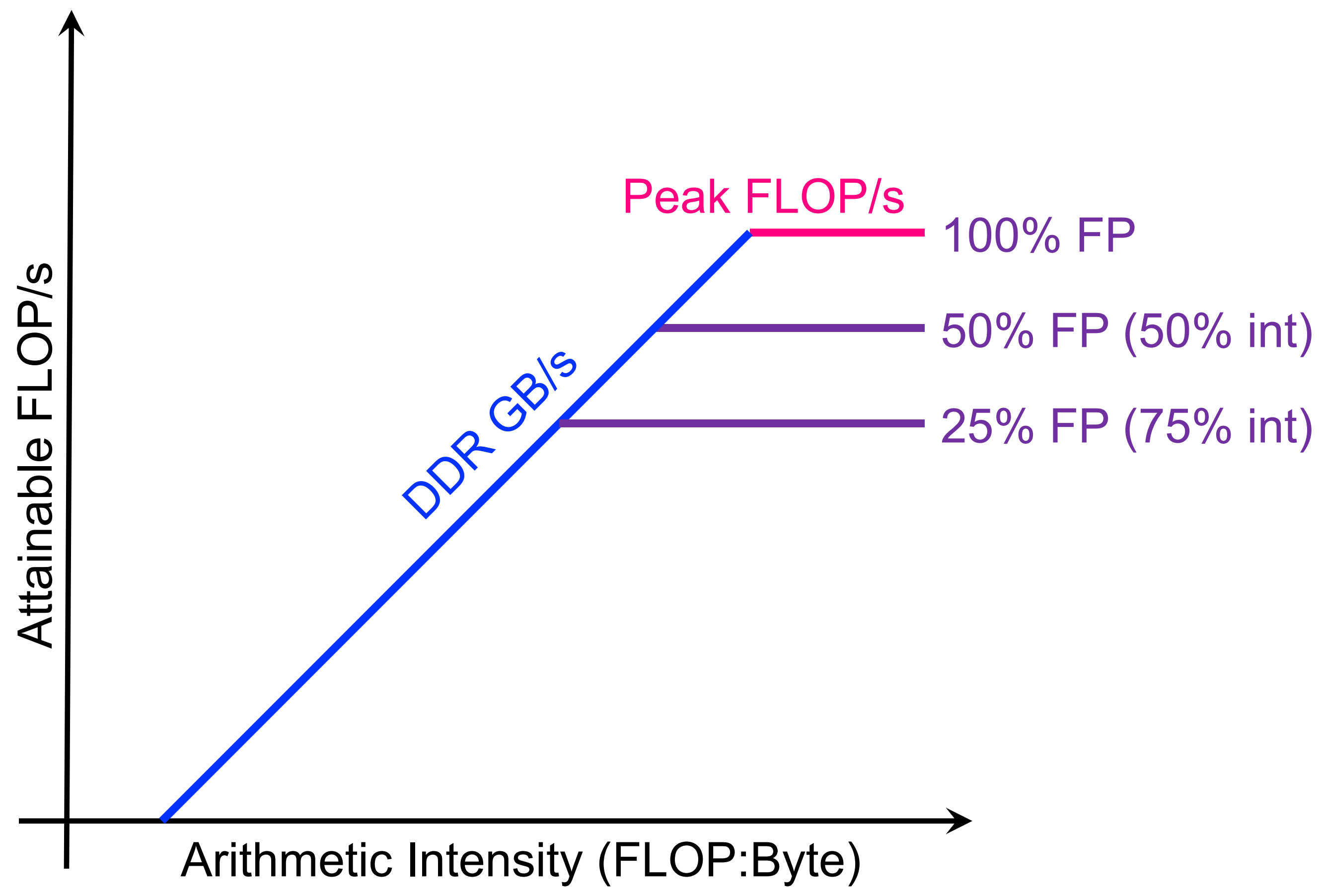
Superscalar vs. Instruction mix

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



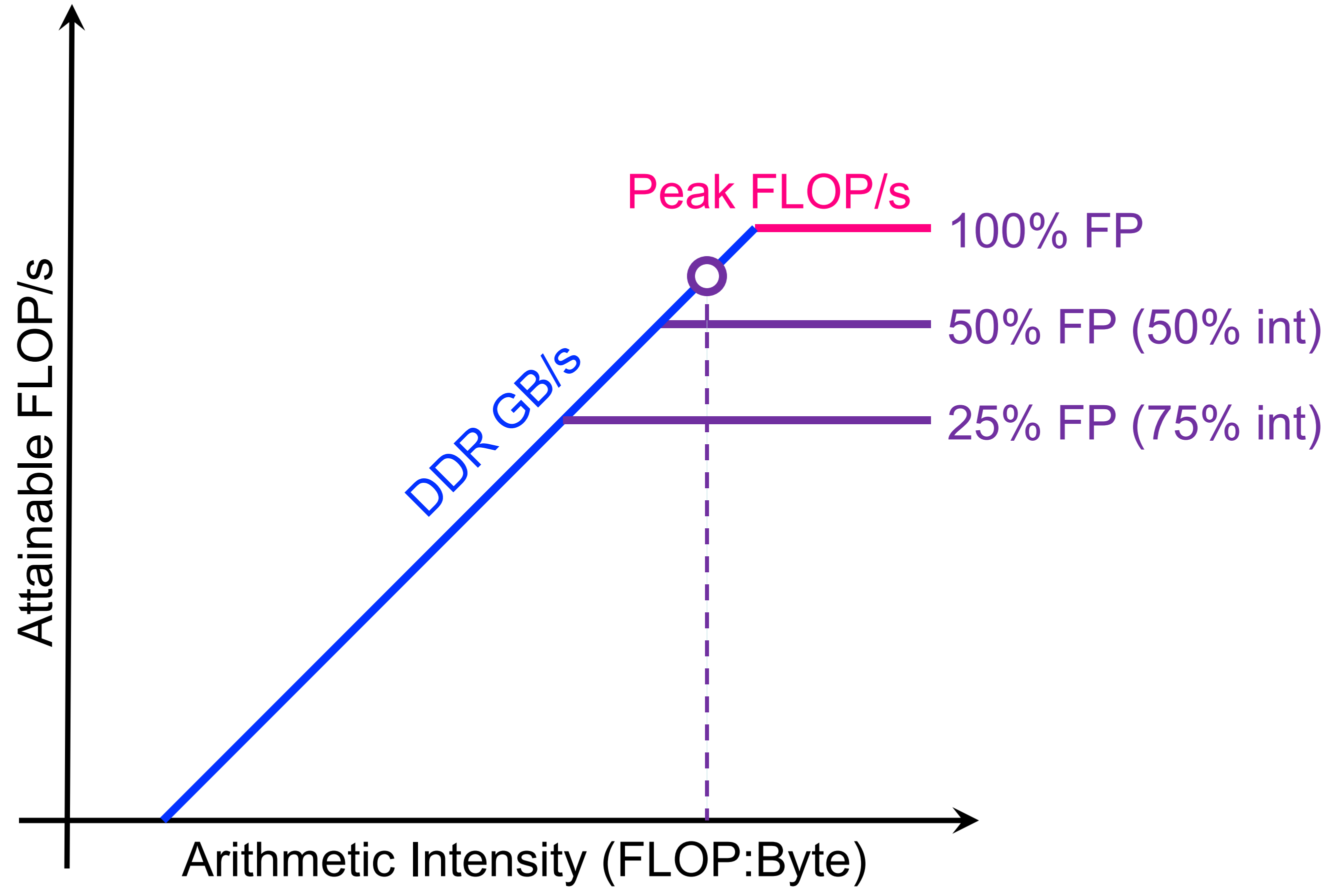
Superscalar vs. Instruction mix

- 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



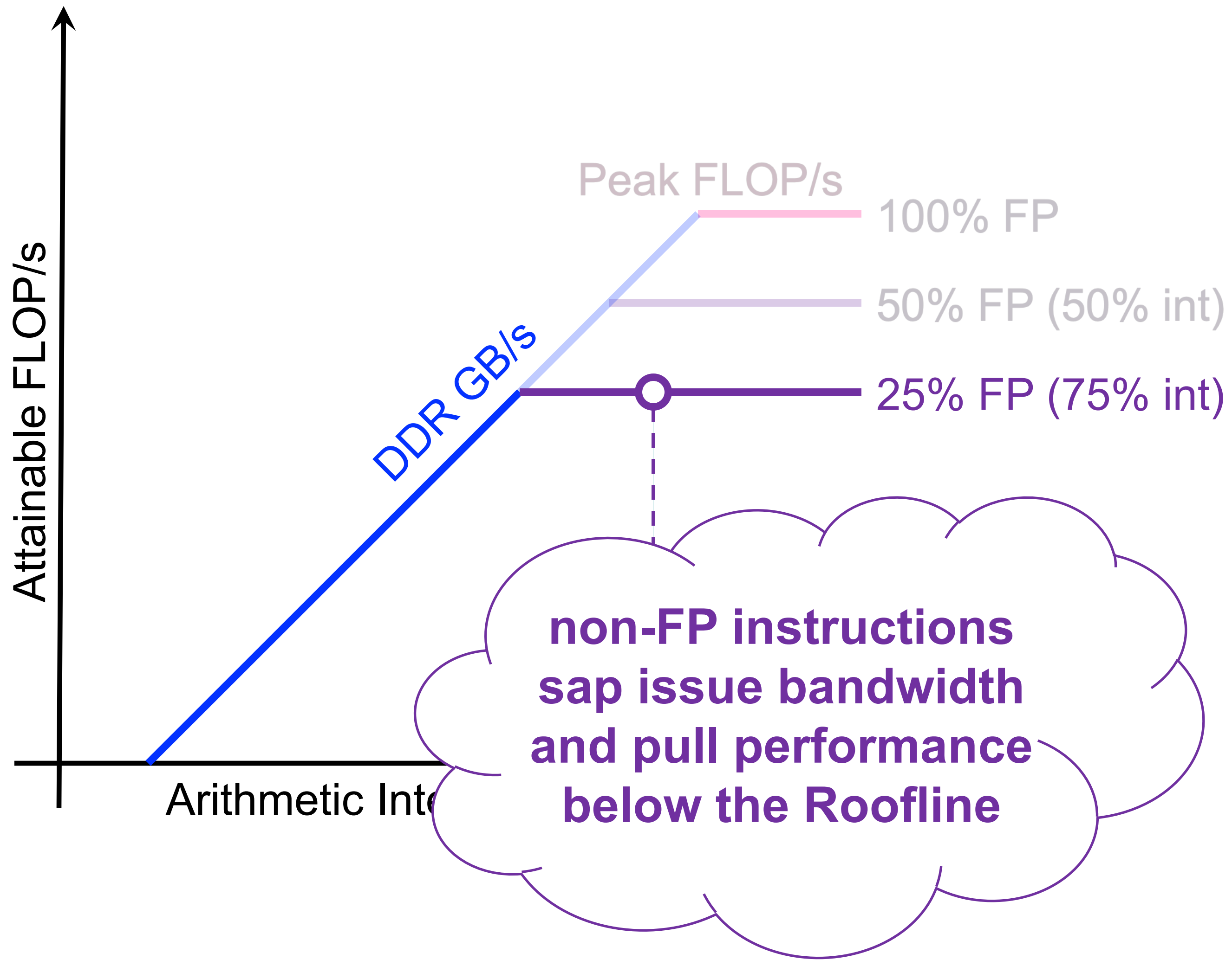
Superscalar vs. Instruction mix

- 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



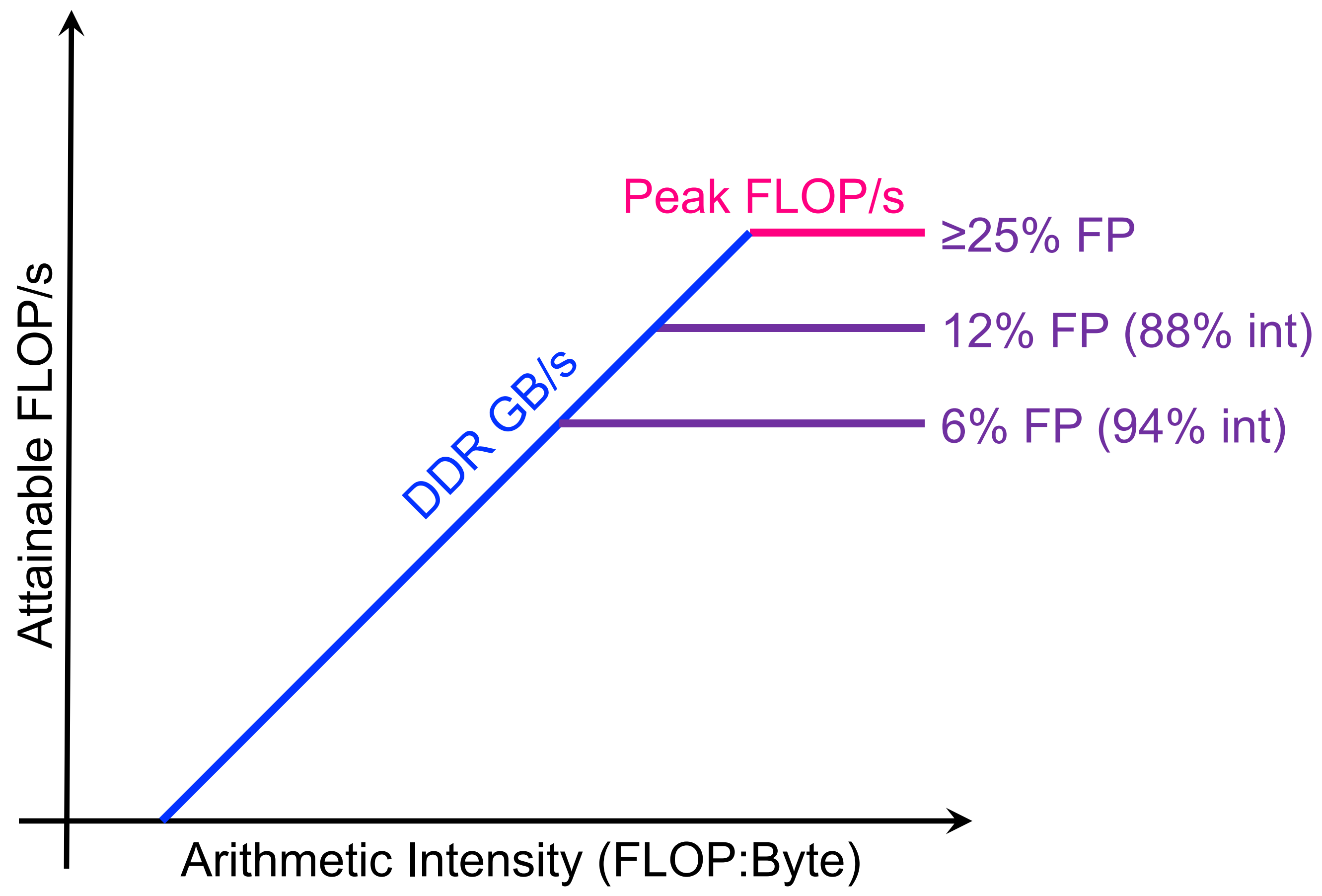
Superscalar vs. Instruction mix

- 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.



BERKELEY LAB

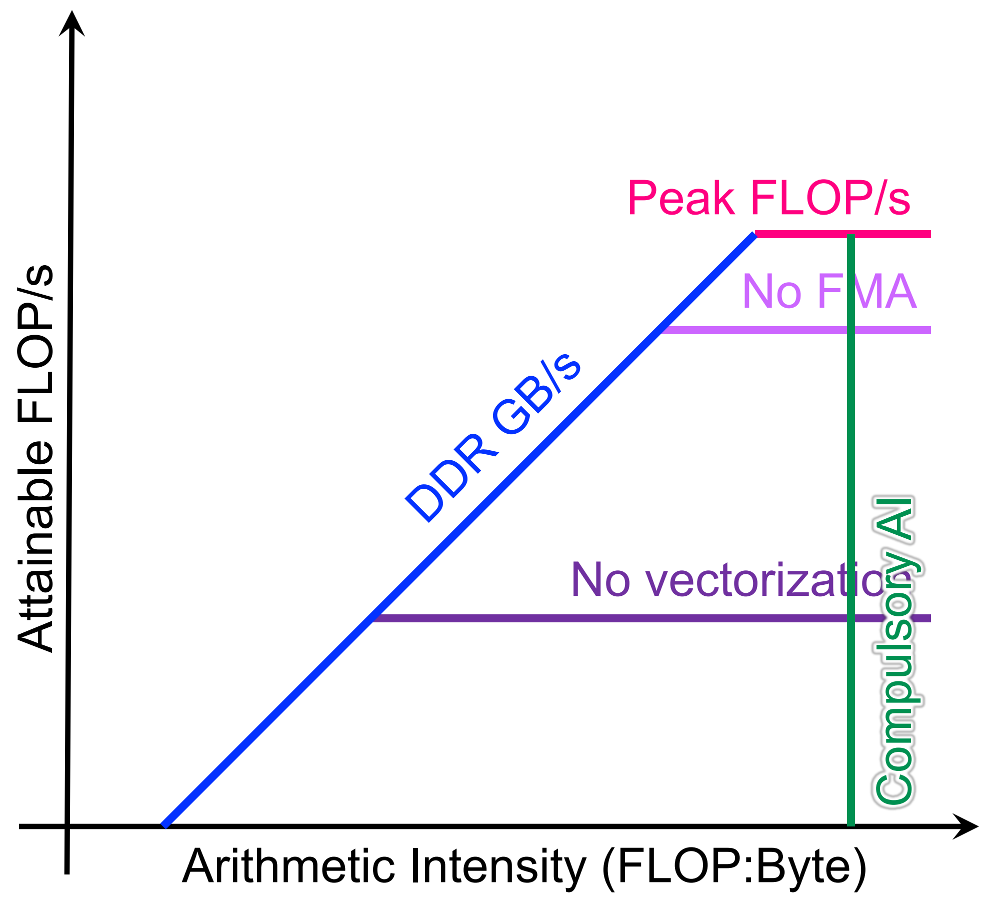
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Extending the Roofline: Modeling Cache Effects

Locality Walls

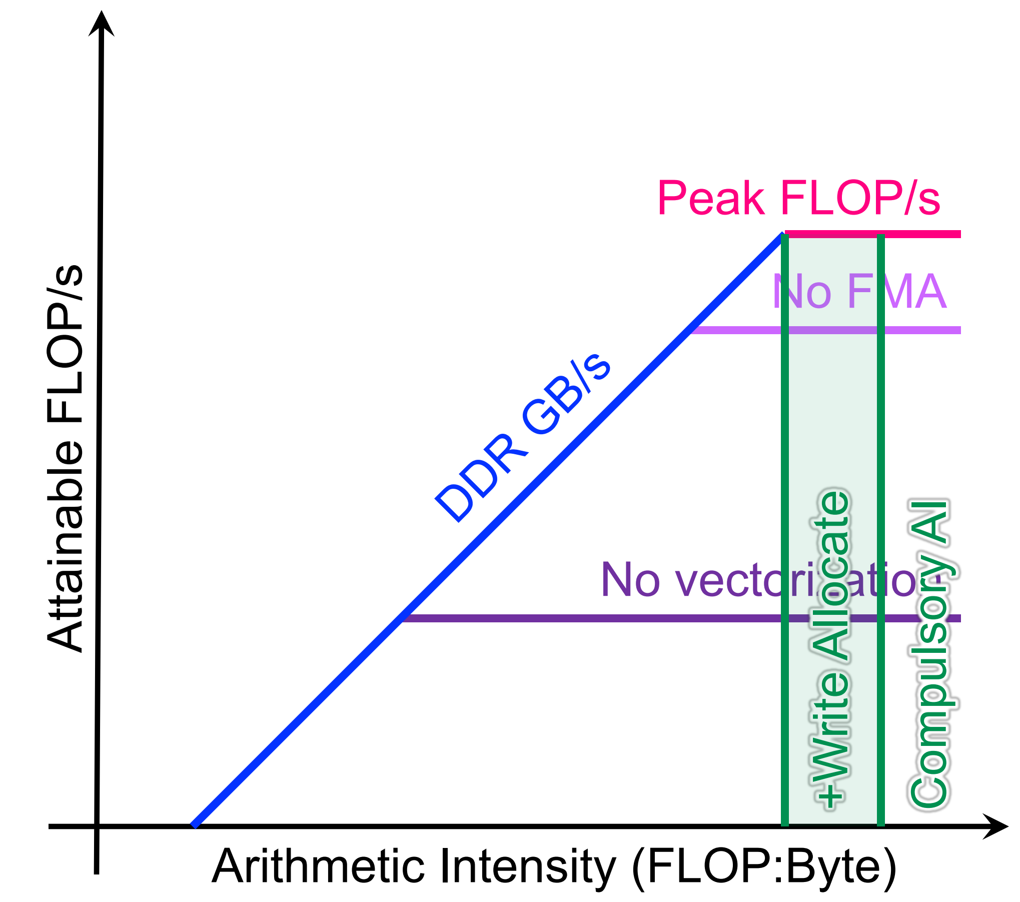
- Naively, we can bound AI using only compulsory cache misses



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

Locality Walls

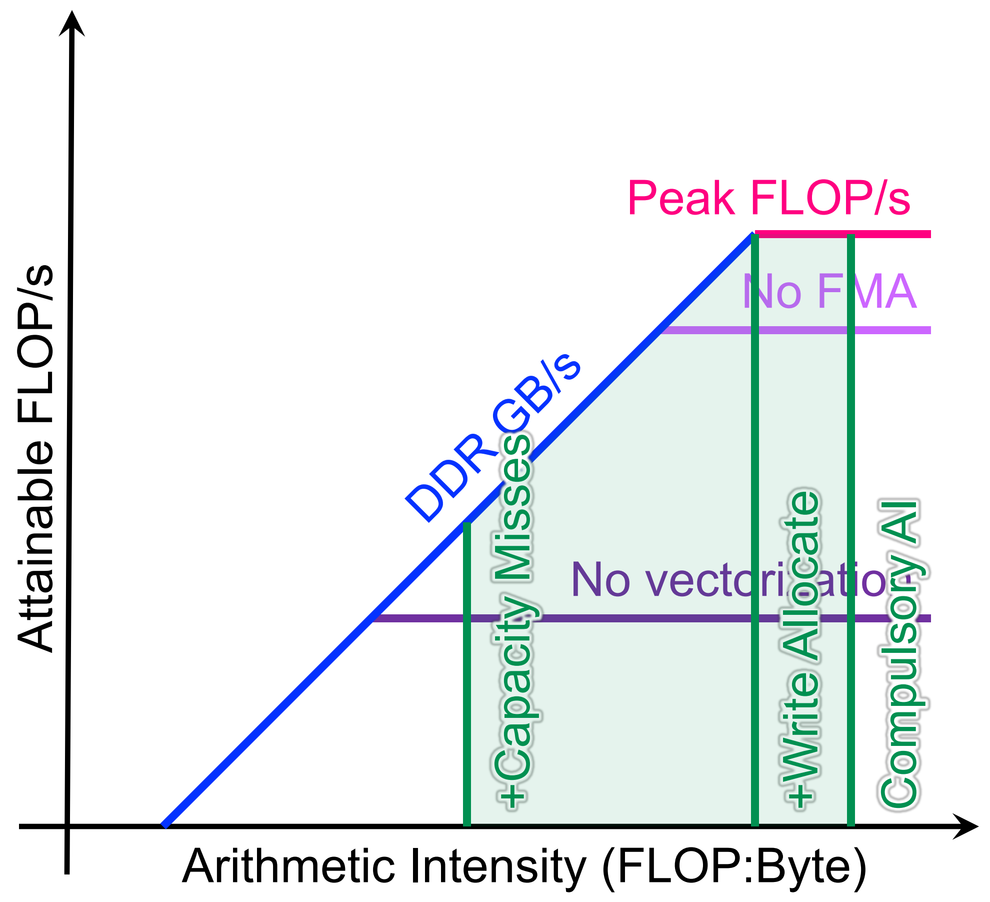
- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI



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

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



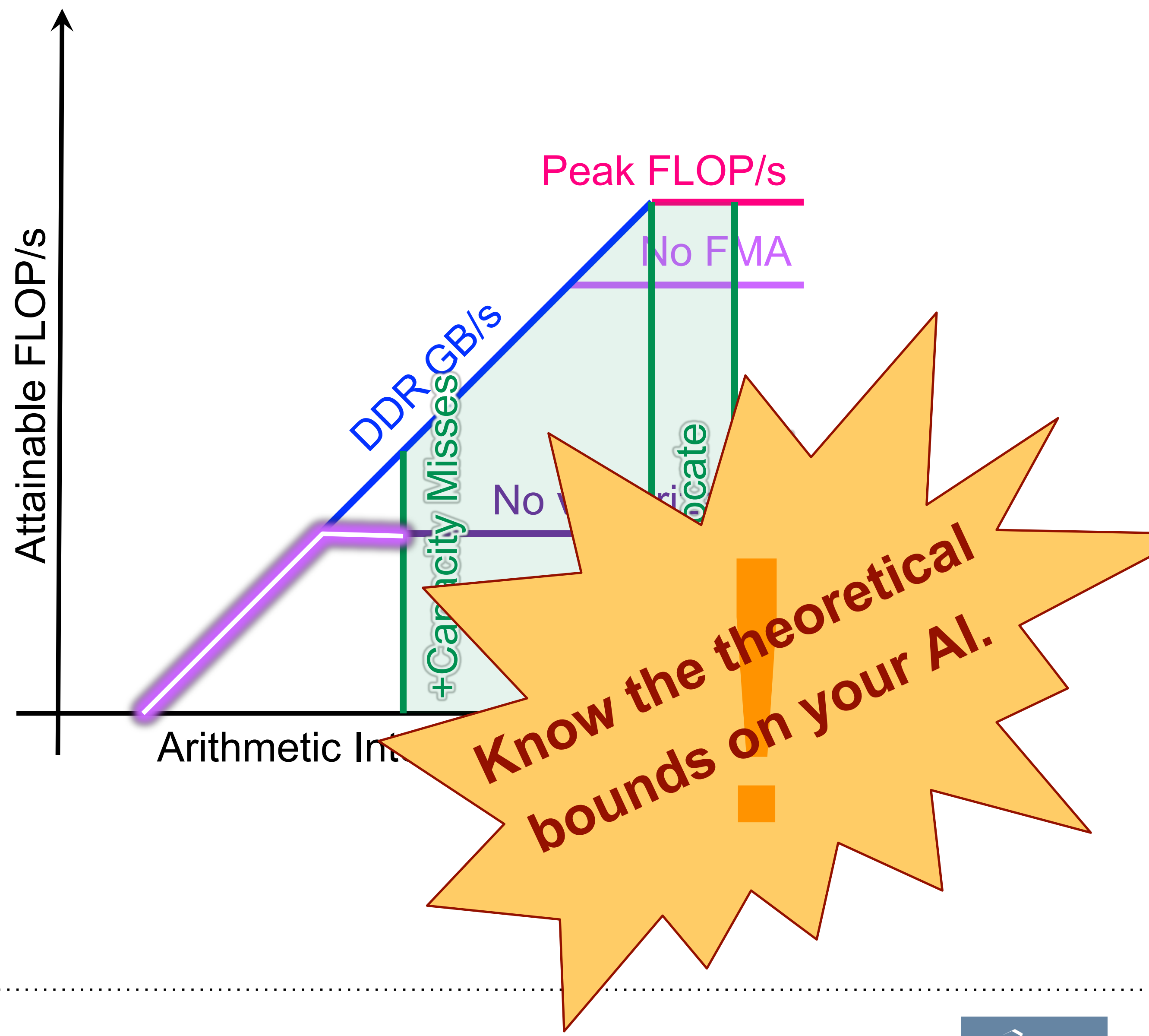
$$AI = \frac{\#FLOPs}{\text{Compulsory Misses + Write Allocates + Capacity Misses}}$$

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**

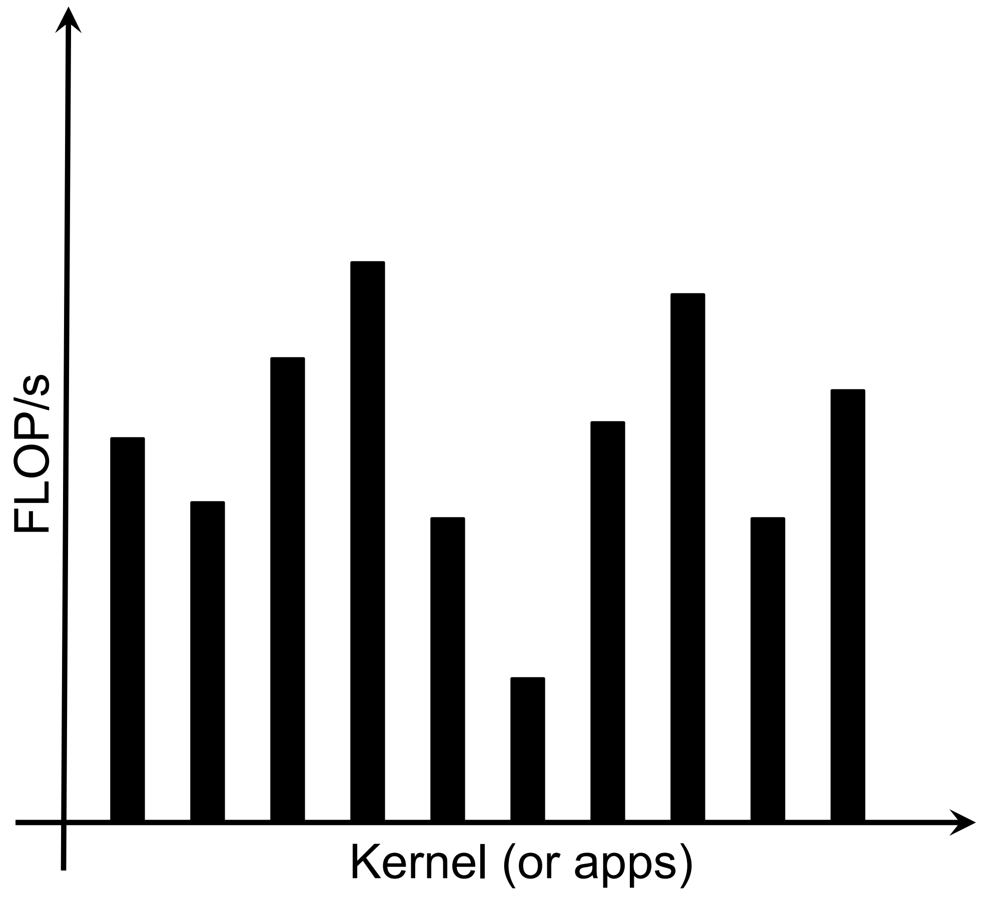
$$AI = \frac{\#FLOPs}{\text{Compulsory Misses} + \text{Write Allocates} + \text{Capacity Misses}}$$



So Why is Roofline Useful?

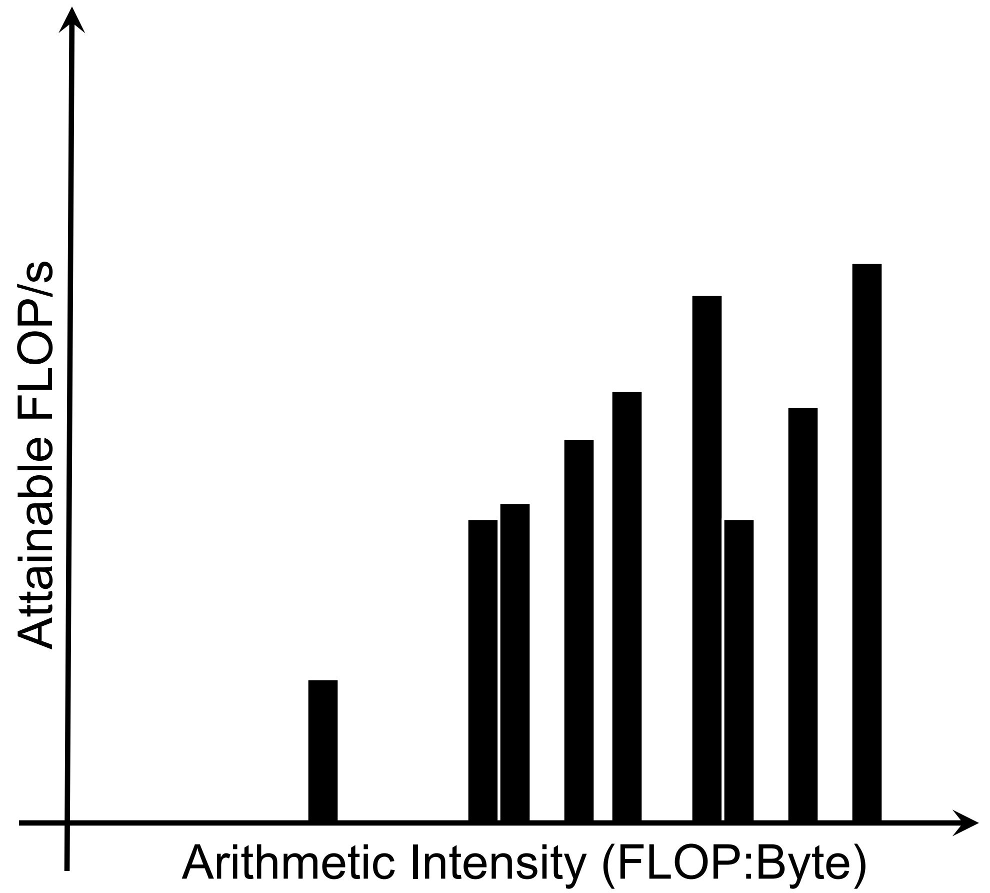
Why is Roofline Useful?

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



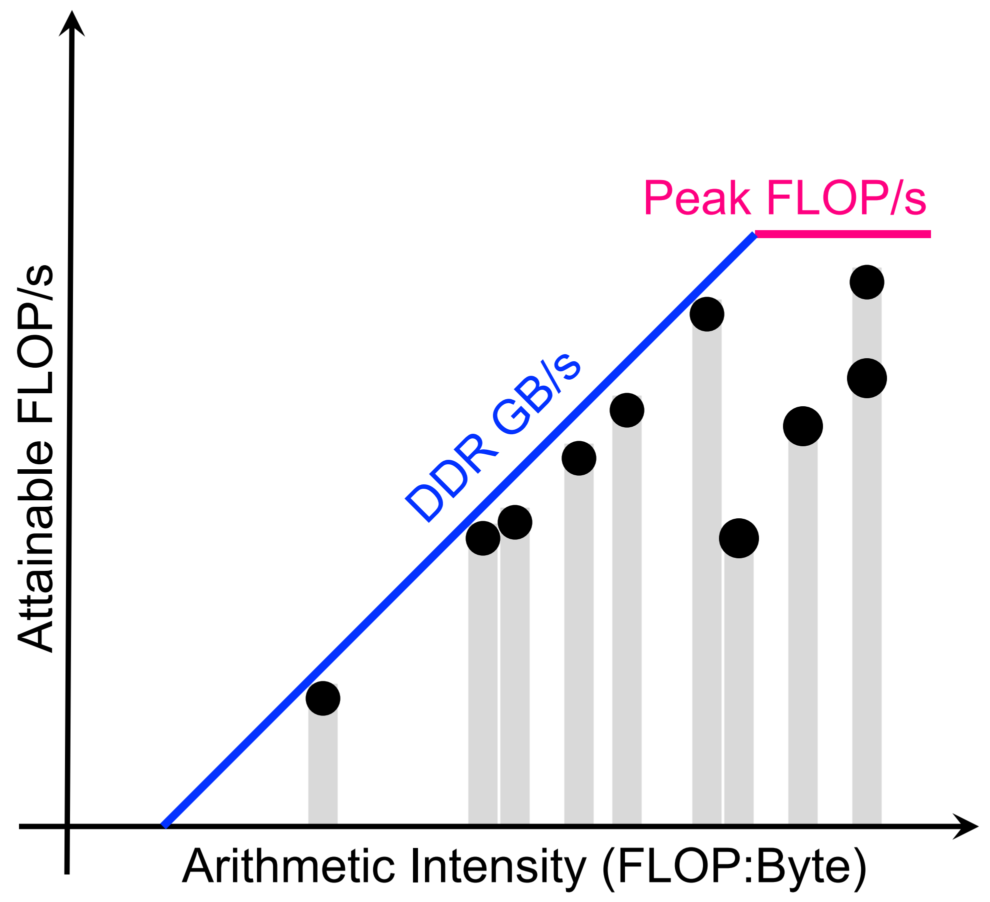
Why is Roofline Useful?

- We can sort kernels by AI …



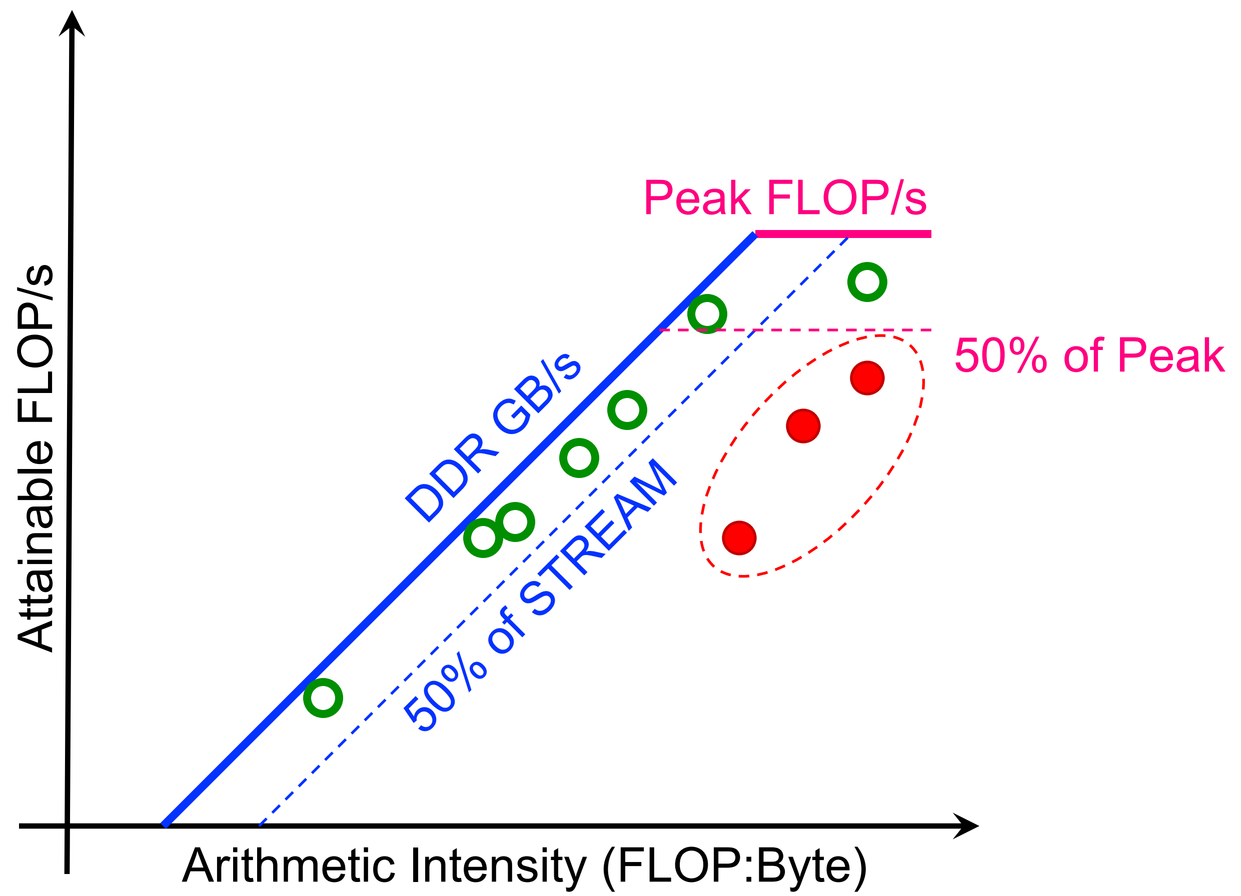
Why is Roofline Useful?

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



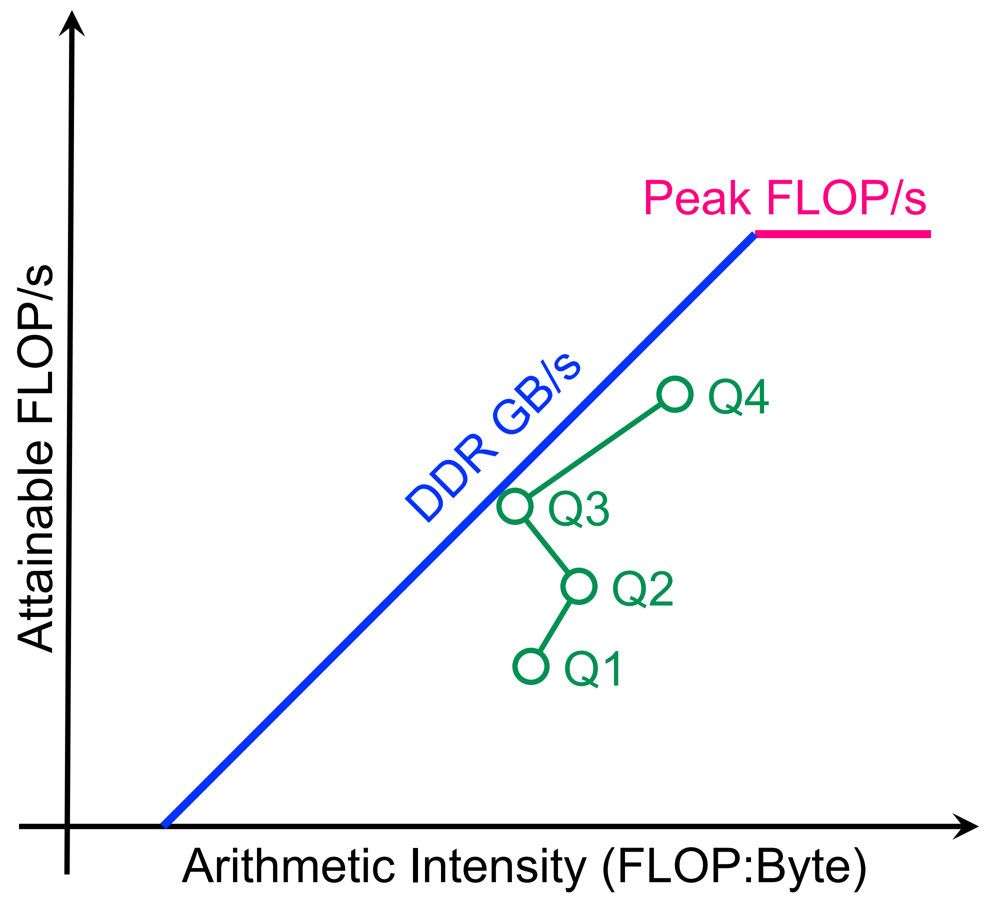
Why is Roofline Useful?

- 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



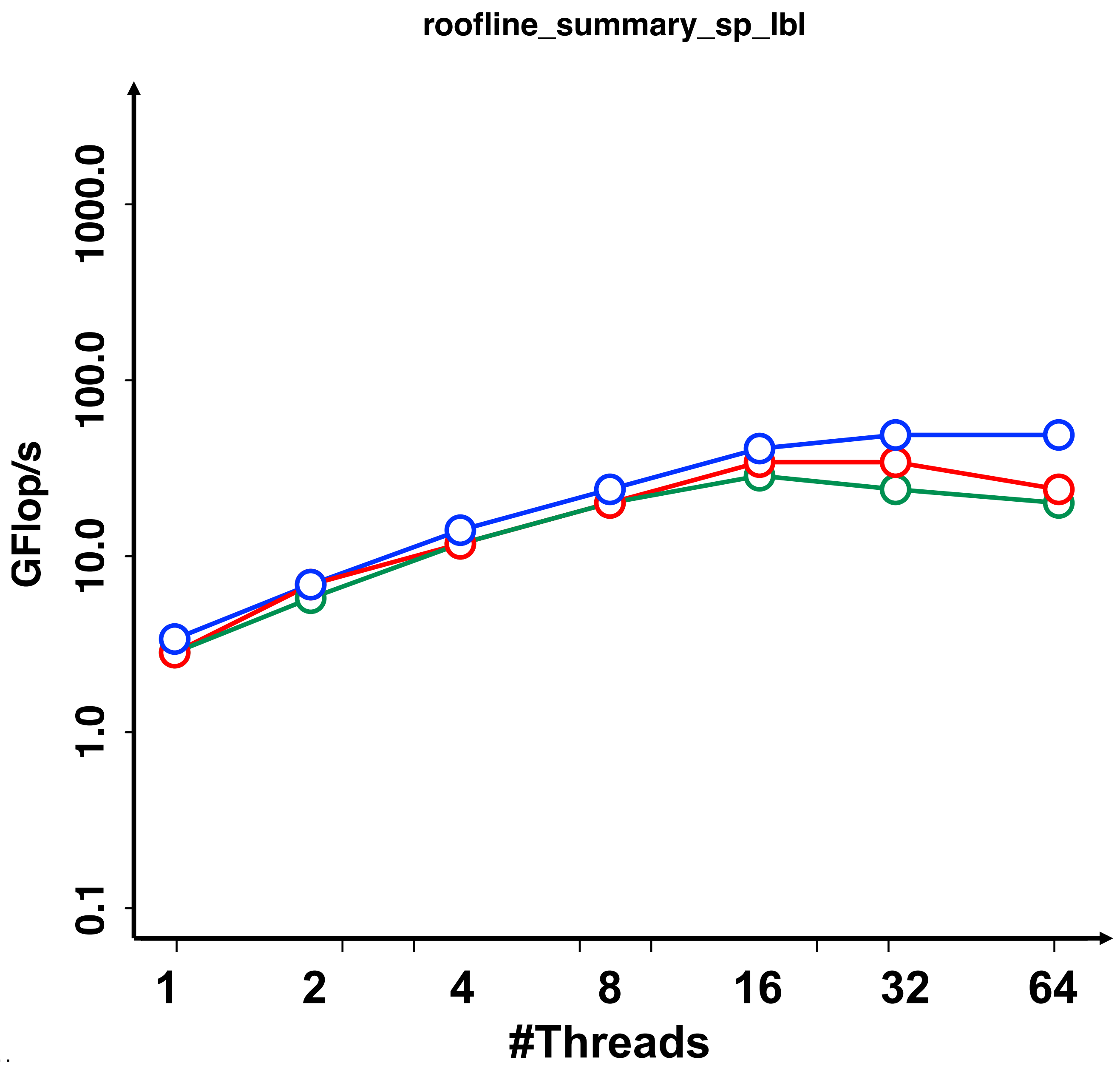
Tracking Progress Towards Optimality

- 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.



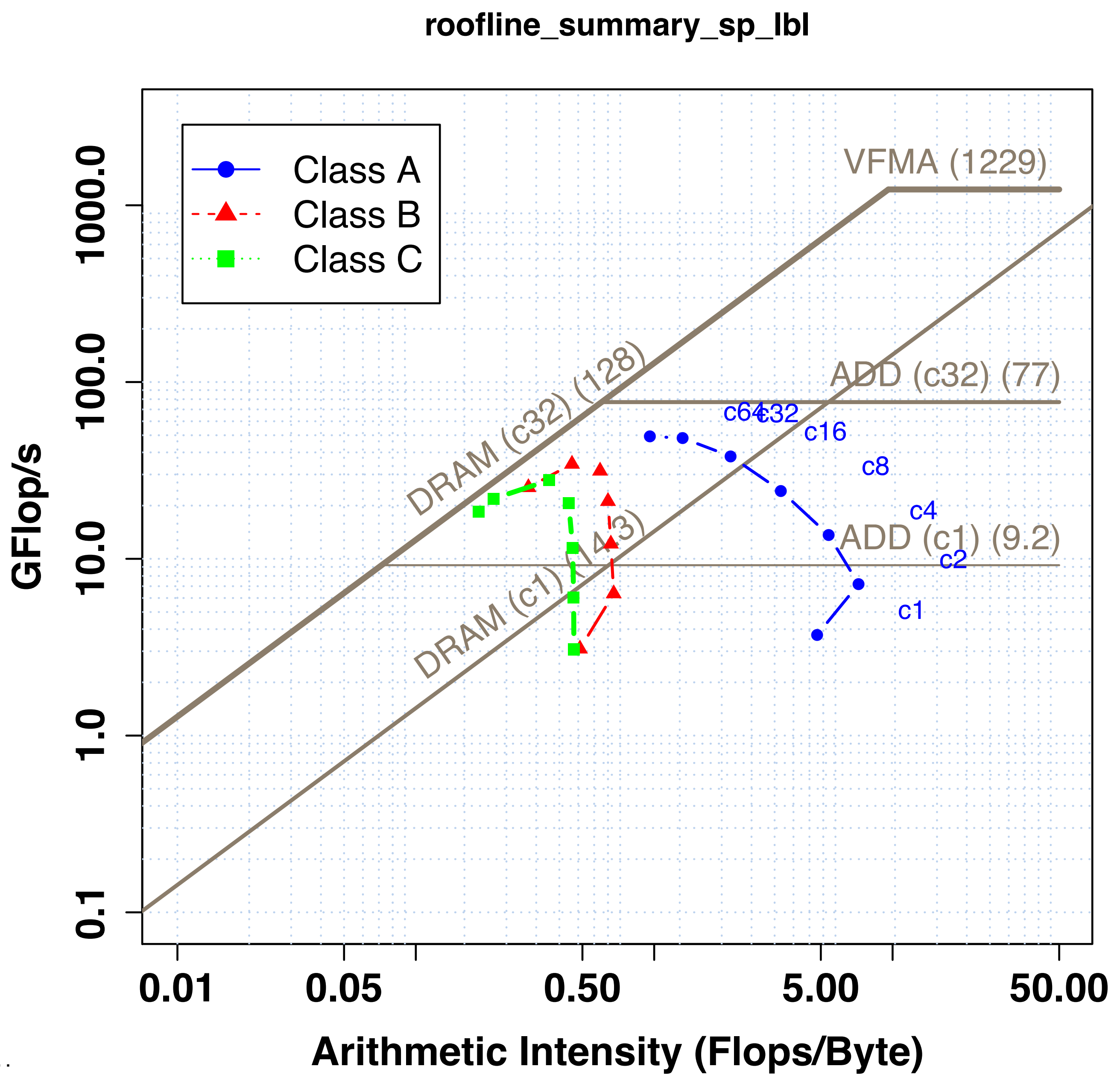
Roofline Scaling Trajectories

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



Roofline Scaling Trajectories

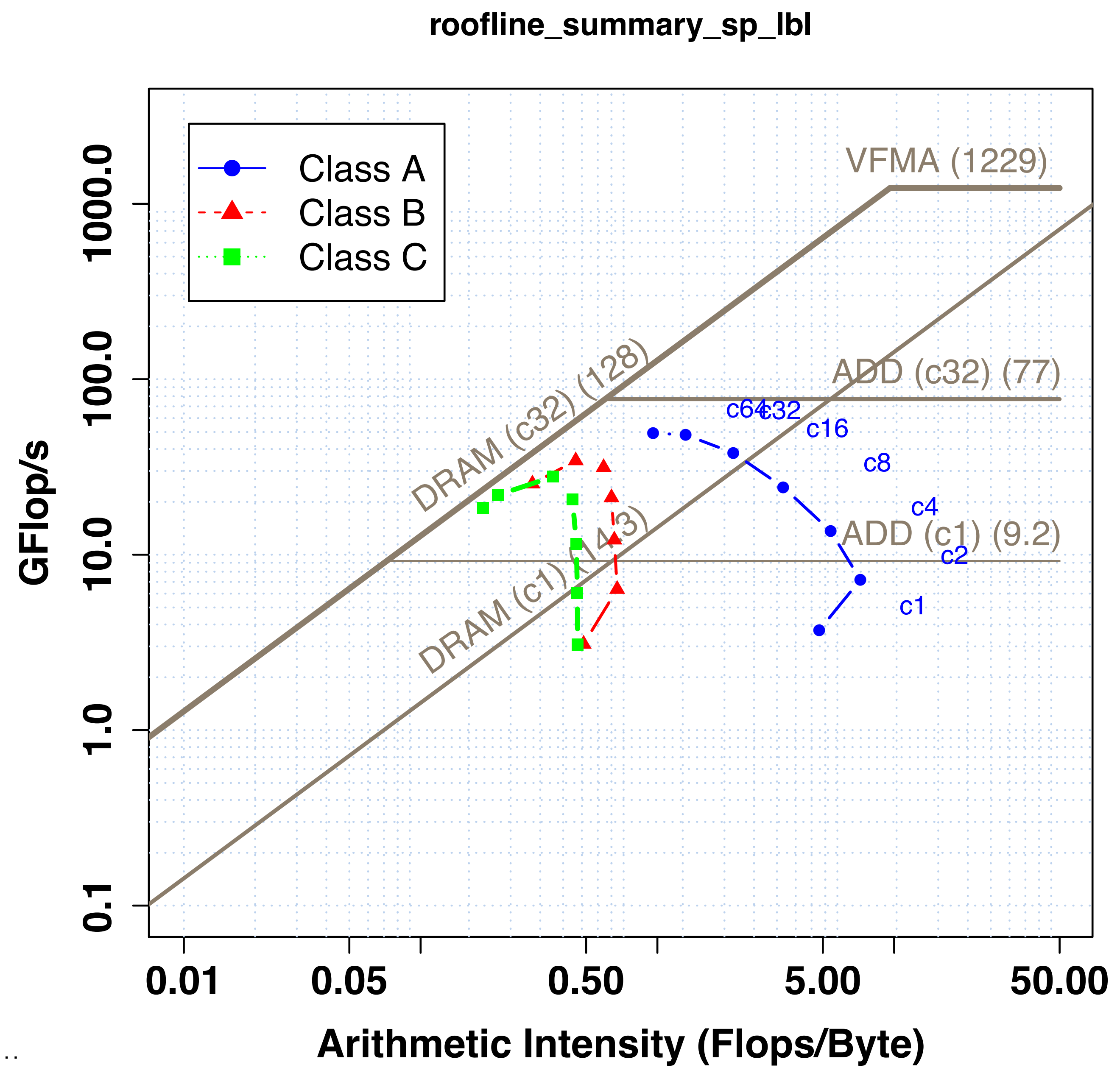
- Often, one plots performance as a function of thread concurrency
  - Carries no insight or analysis
  - Provides no actionable information.
- Khaled Ibrahim developed a new way of using Roofline to analyze thread (or process) scalability
  - Create a 2D scatter plot of performance as a function of 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.

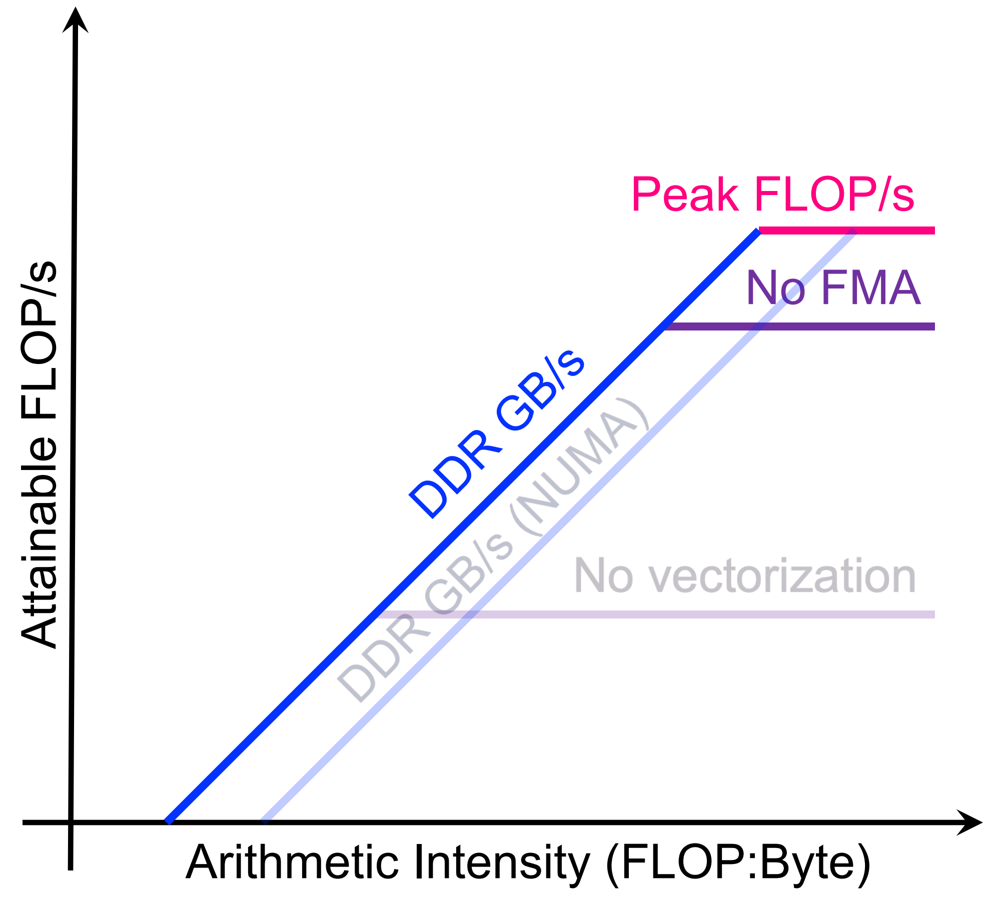
Roofline Scaling Trajectories

- 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



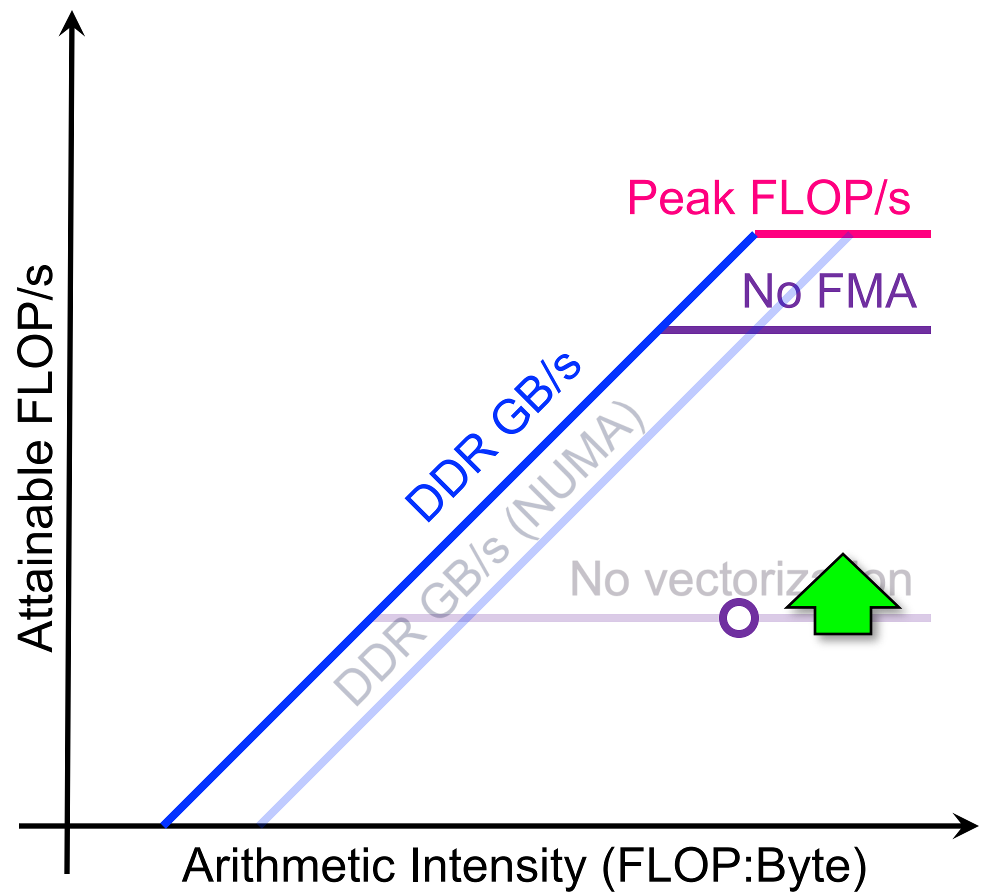
Driving Performance Optimization

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



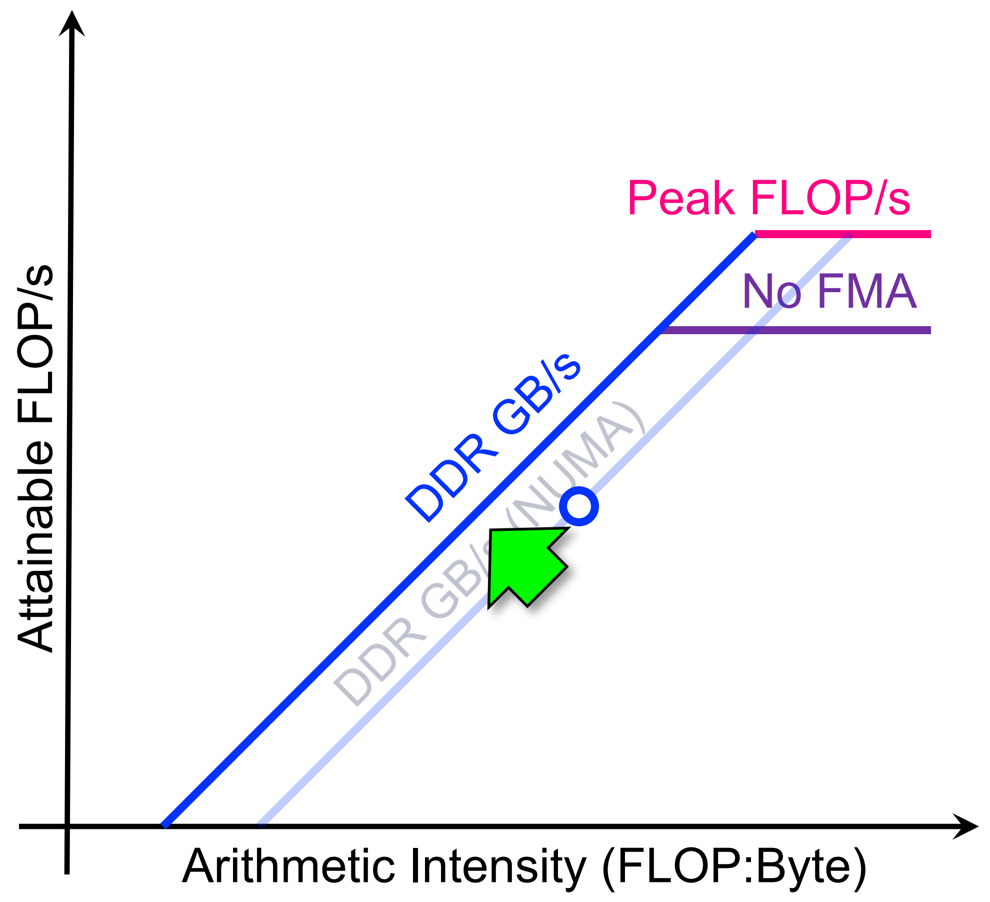
Driving Performance Optimization

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



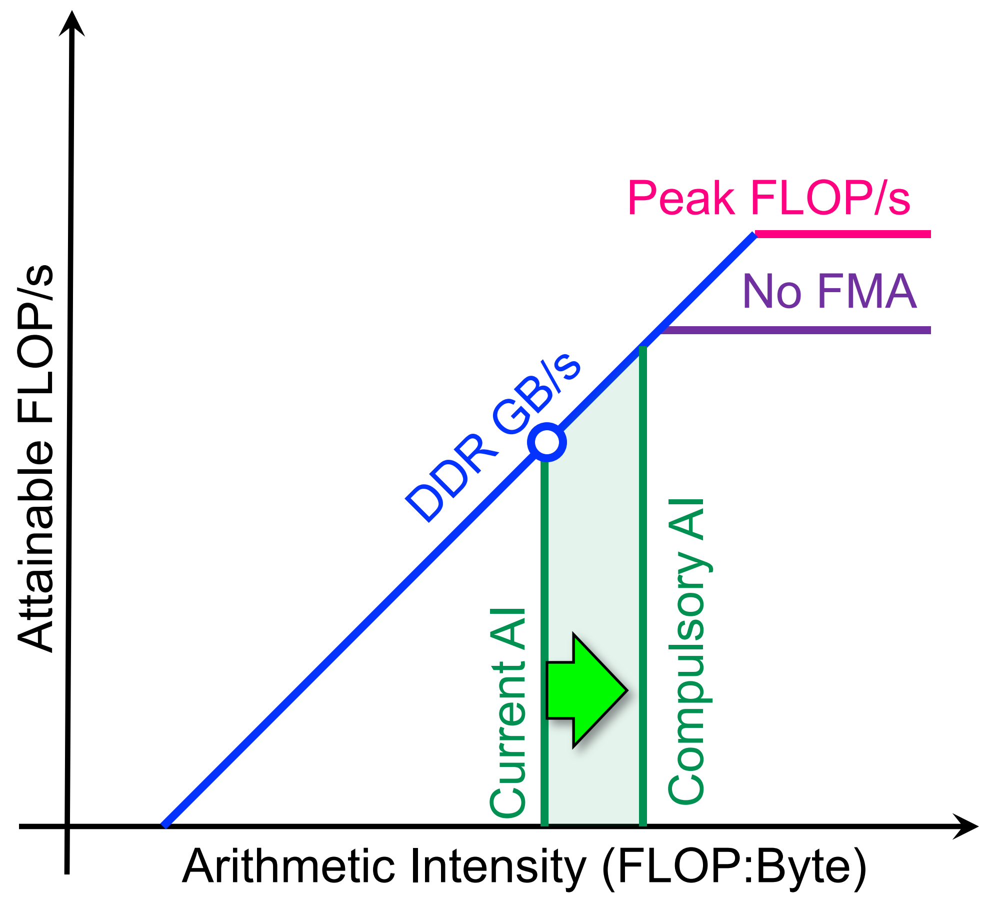
Driving Performance Optimization

- 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)**



BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

How do I build and use 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.

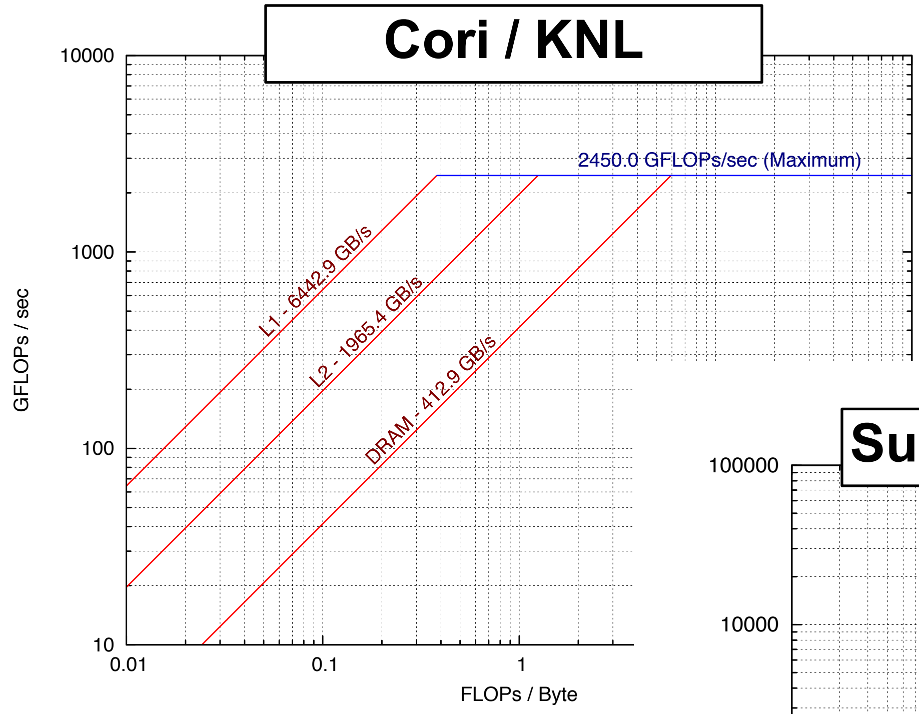
➢ **Take marketing numbers with a grain of salt**

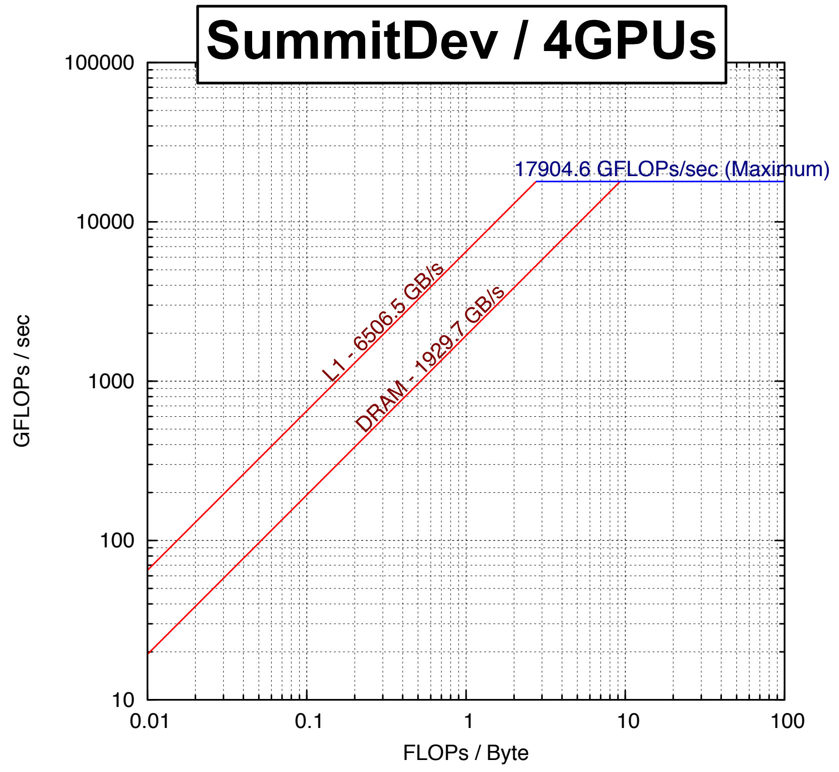
Machine Characterization

- 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.
- 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://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

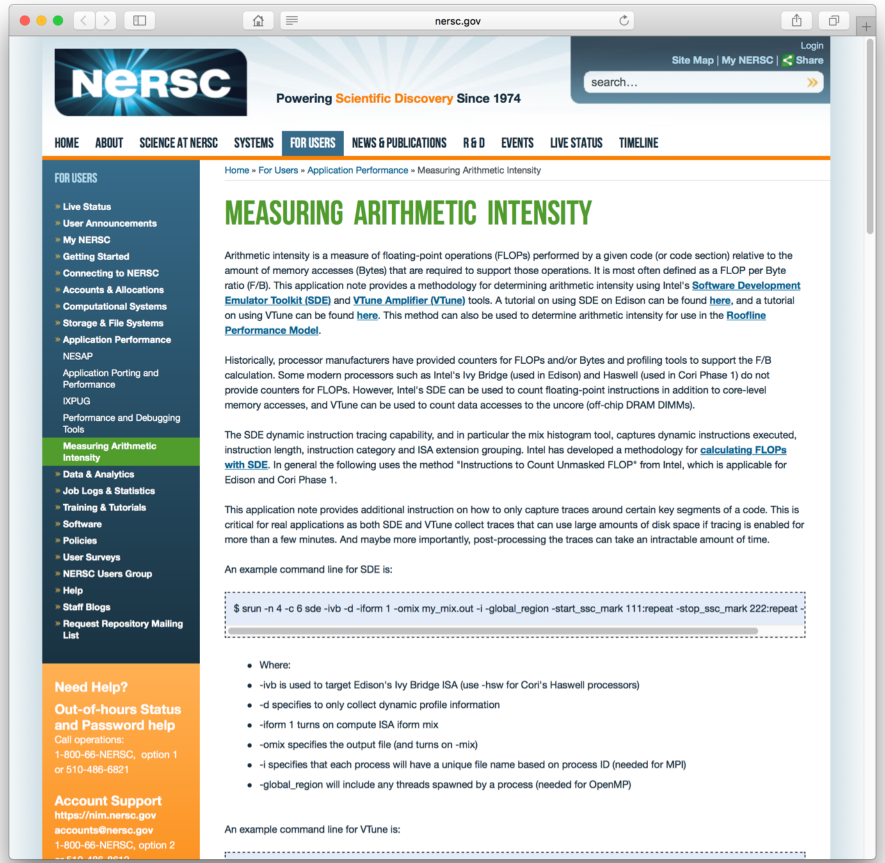
- ▪ Read counter before/after
- ✓ Applies to full hierarchy (L2, DRAM,
- ✓ Much more Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✗ Requires privileged access
- ✗ Requires manual instrumentation (+overhead) or full-app characterization

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…



Need Help?

Out-of-hours Status and Password help
Call operations:
1-800-66-NERSC, option 1
or 510-486-6821

Account Support
https://nim.nersc.gov
accounts@nersc.gov
1-800-66-NERSC, option 2

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:

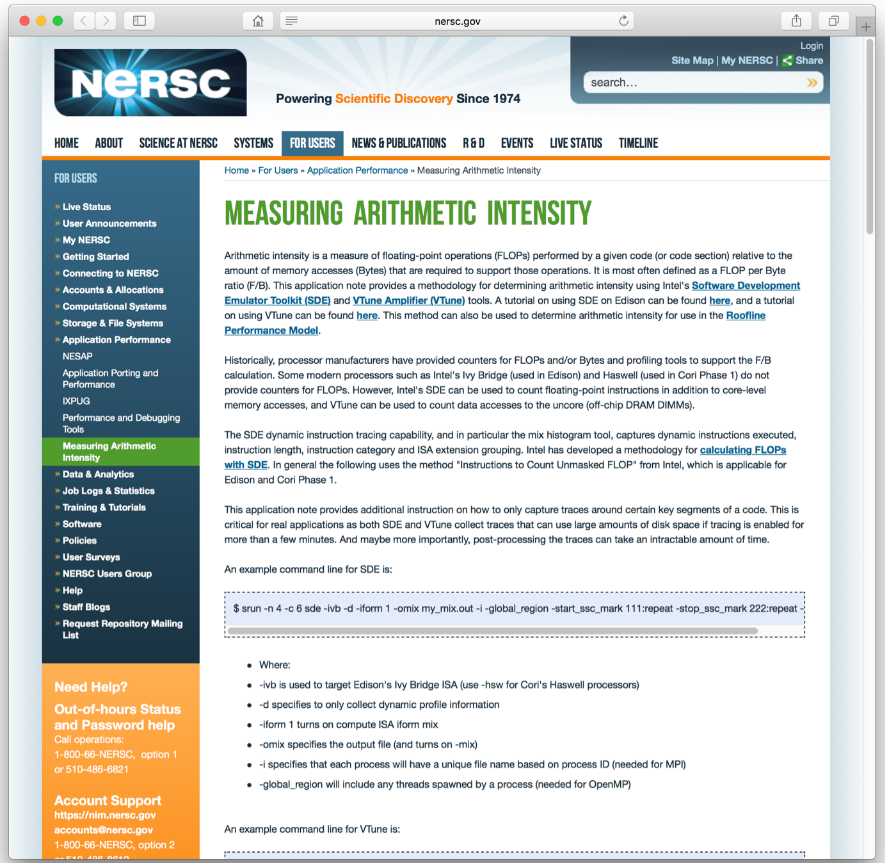
http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

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)

BERKELEY LAB

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…



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)

BERKELEY LAB

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

Profiling with LIKWID

- likwid-perfctr (threaded) + likwid-mpirun (MPI/hybrid)

- no GUI
- low overhead -> SDE, VTune, etc
- no code instrumentation required -> CrayPat-tracing
- no root access required -> VTune
- no extra modules required to be installed -> VTune

- use Linux 'msr' module to access MSR (Model Specific Register) files

- Cori:

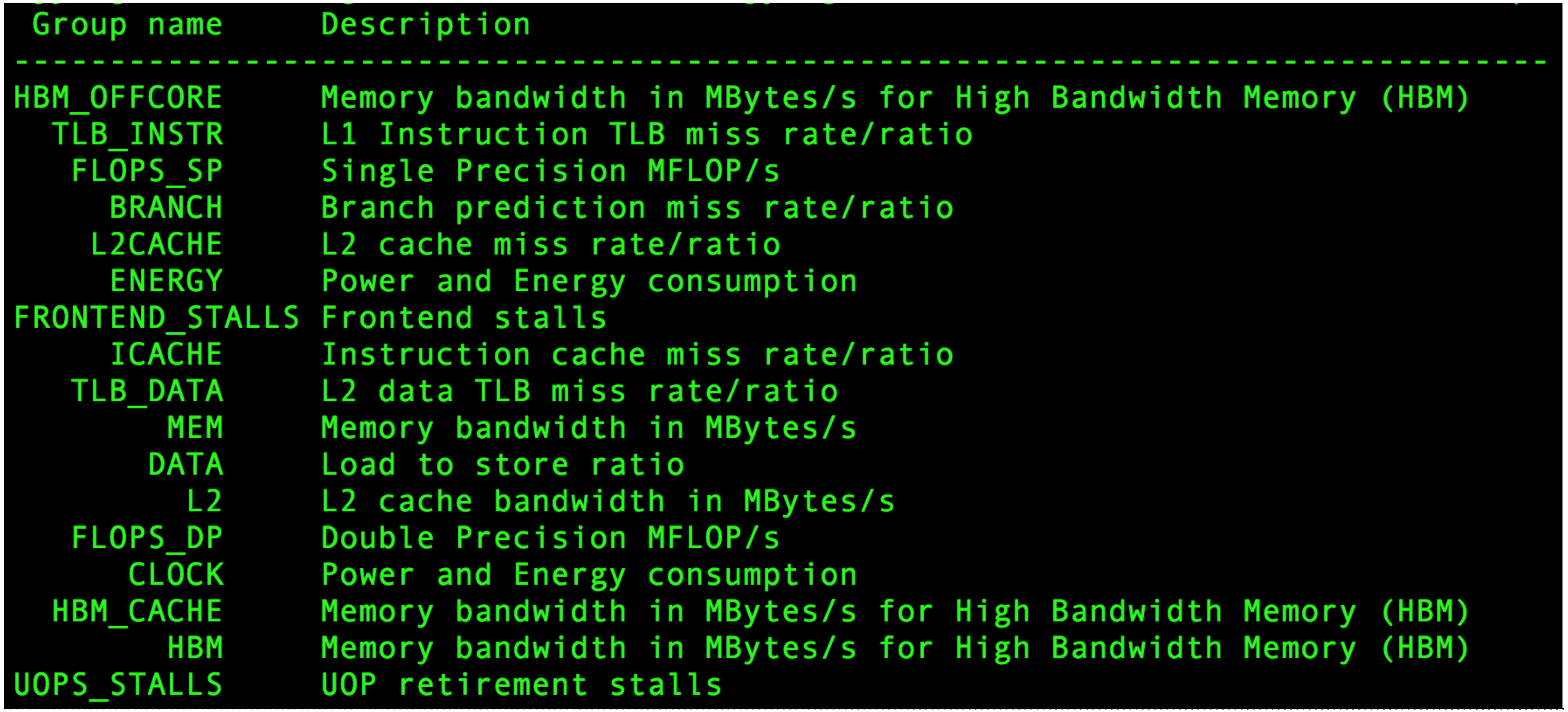
```
module load vtune
sbatch/salloc --perf=likwid
module load likwid
```

Profiling with LIKWID (2)

- Alternately, one can construct a script and monitor only process 0

```
srun -n8 -c32              ./a.out args
srun -n8 -c32 ./perfctr.sh ./a.out args

where perfctr.sh is
#!/bin/bash
let SLURM_MPI_RANK=$SLURM_PROCID
if [ $SLURM_MPI_RANK = 0 ];then
# only process 0 runs likwid and it monitors only logical CPUs 0-31
likwid-perfctr -C 0-31 -g CACHES $@
else
$@
fi
```

Likwid-perfctr –a (KNL)



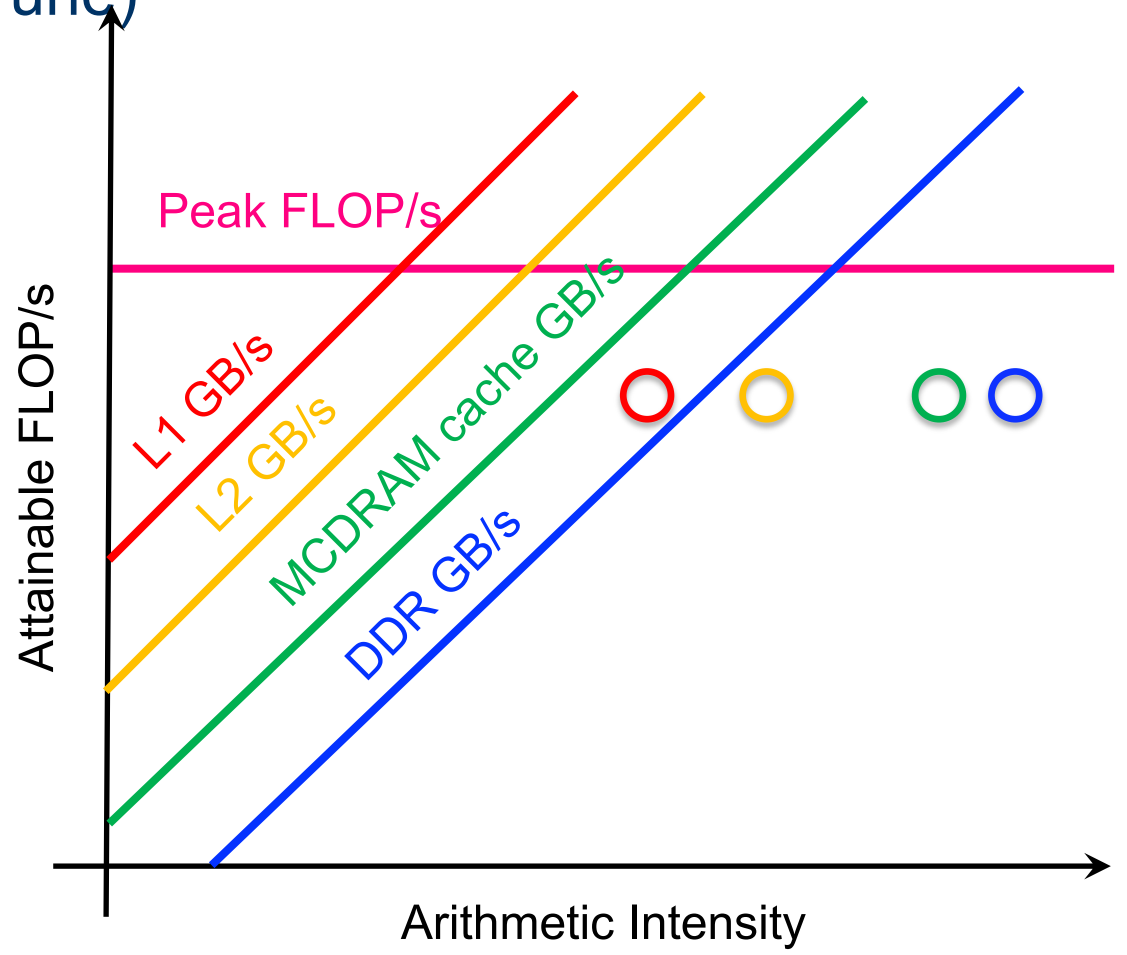
```
 Group name     Description
--------------------------------------------------------------------------------
HBM_OFFCORE     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
  TLB_INSTR     L1 Instruction TLB miss rate/ratio
   FLOPS_SP     Single Precision MFLOP/s
     BRANCH     Branch prediction miss rate/ratio
    L2CACHE     L2 cache miss rate/ratio
     ENERGY     Power and Energy consumption
FRONTEND_STALLS Frontend stalls
     ICACHE     Instruction cache miss rate/ratio
   TLB_DATA     L2 data TLB miss rate/ratio
        MEM     Memory bandwidth in MBytes/s
       DATA     Load to store ratio
         L2     L2 cache bandwidth in MBytes/s
   FLOPS_DP     Double Precision MFLOP/s
      CLOCK     Power and Energy consumption
  HBM_CACHE     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
        HBM     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
UOPS_STALLS     UOP retirement stalls
```

Using LIKWID for Roofline

- GPP kernel from BerkeleyGW
- Arithmetic Intensity = FLOPS / Bytes (= SDE / VTune)
  = FLOPS/sec / Bytes/sec
  = **FLOPS_DP / Bandwidth**

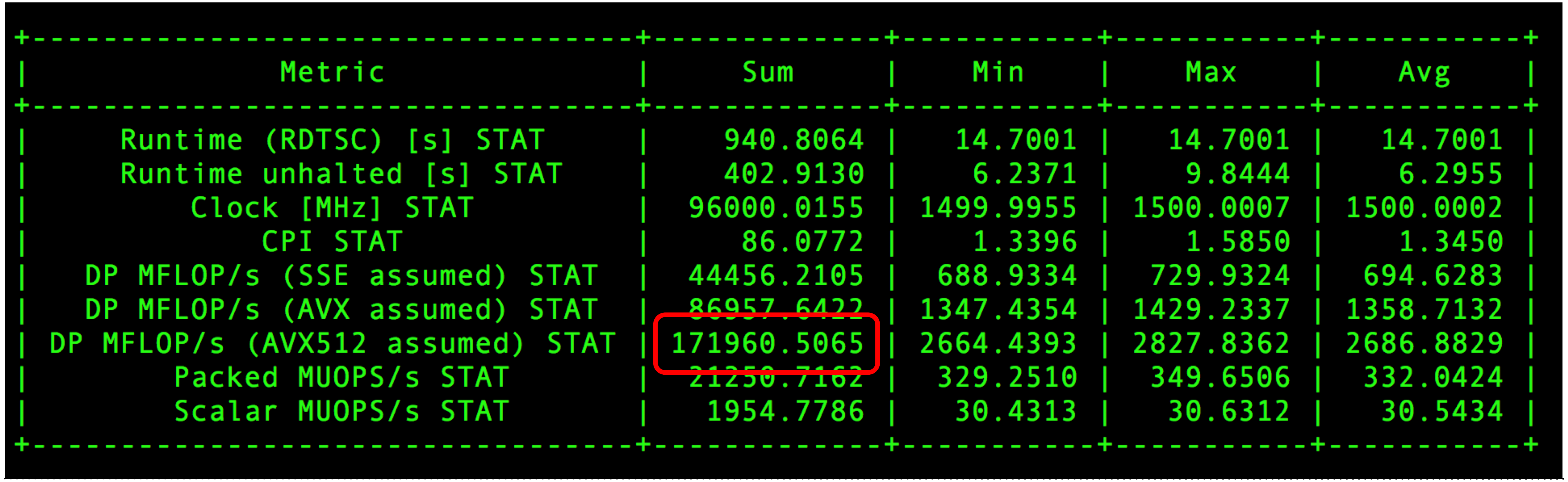
- AI (DRAM) = FLOPS_DP / Bandwidth (DRAM)
- AI (MCDRAM) = FLOPS_DP / Bandwidth (MCDRAM)
- AI (L2) = FLOPS_DP / Bandwidth (L2)
- AI (L1) = FLOPS_DP / Bandwidth (L1)

- **Performance = FLOPS_DP**



GFLOP/s

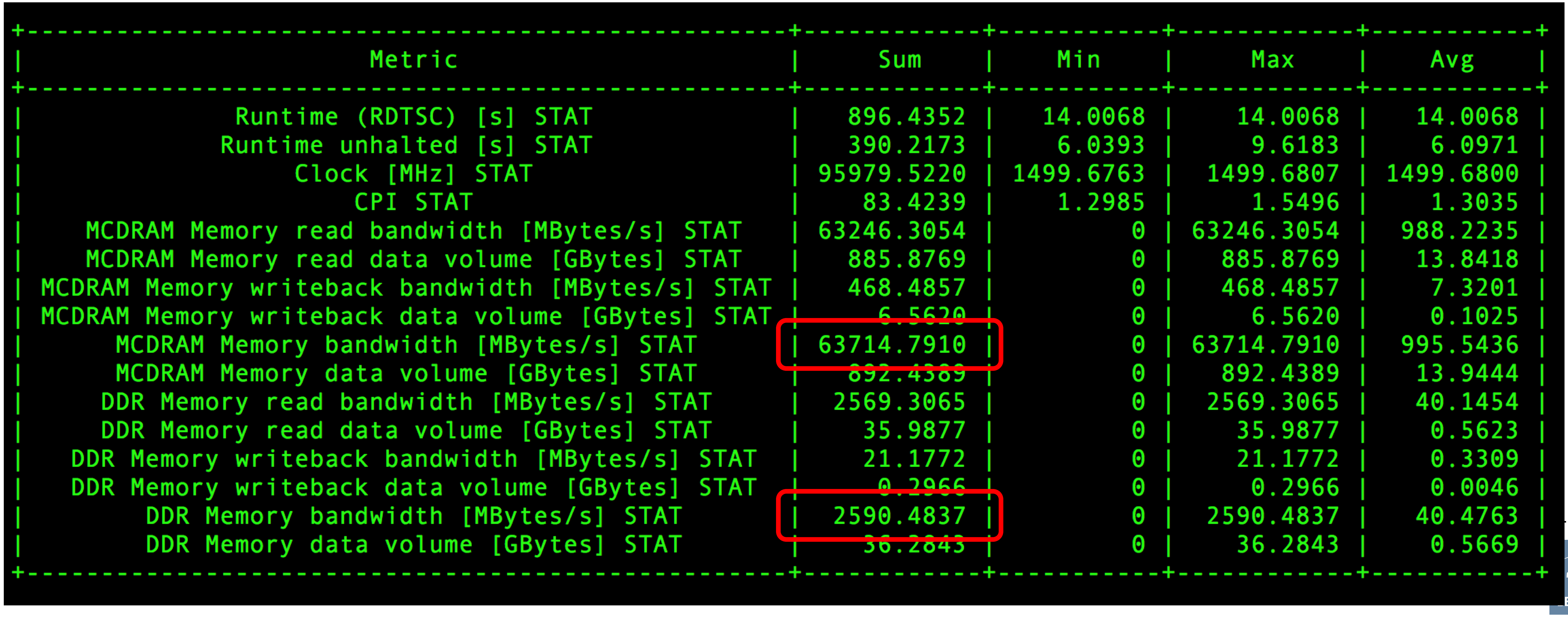
- GPP kernel on KNL: **171.960 GFLOPS/sec**
  - UOPS_RETIRED_PACKED_SIMD
  - UOPS_RETIRED_SCALAR_SIMD
- likwid-perfctr -C 0-63 -g **FLOPS_DP** ./gpp.knl.ex 512 2 32768 20
  - 8*UOPS_RETIRED_PACKED_SIMD+UOPS_RETIRED_SCALAR_SIMD



```
+-----------------------------------+-------------+------------+------------+------------+
|              Metric               |     Sum     |    Min     |    Max     |    Avg     |
+-----------------------------------+-------------+------------+------------+------------+
|      Runtime (RDTSC) [s] STAT     |    940.8064 |    14.7001 |    14.7001 |    14.7001 |
|      Runtime unhalted [s] STAT    |    402.9130 |     6.2371 |     9.8444 |     6.2955 |
|          Clock [MHz] STAT         |  96000.0155 |  1499.9955 |  1500.0007 |  1500.0002 |
|              CPI STAT             |     86.0772 |     1.3396 |     1.5850 |     1.3450 |
|    DP MFLOP/s (SSE assumed) STAT  |  44456.2105 |   688.9334 |   729.9324 |   694.6283 |
|    DP MFLOP/s (AVX assumed) STAT  |  86957.6422 |  1347.4354 |  1429.2337 |  1358.7132 |
|  DP MFLOP/s (AVX512 assumed) STAT | 171960.5065 |  2664.4393 |  2827.8362 |  2686.8829 |
|        Packed MUOPS/s STAT        |  21250.7162 |   329.2510 |   349.6506 |   332.0424 |
|        Scalar MUOPS/s STAT        |   1954.7786 |    30.4313 |    30.6312 |    30.5434 |
+-----------------------------------+-------------+------------+------------+------------+
```

MCDRAM and DDR GB/s

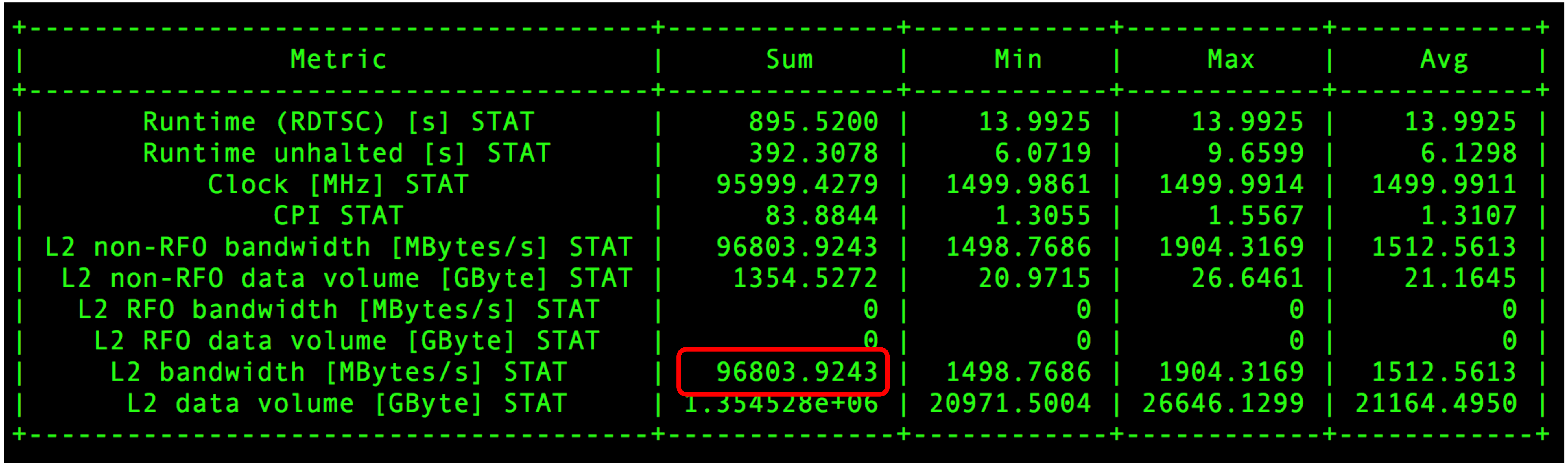
- kernel on KNL: **DDR 2.59GB/s + MCDRAM 63.71GB/s**
  - MC_CAS_READS/ MC_CAS_WRITES
  - EDC_RPQ_INSERTS/ EDC_WPQ_INSERTS
  - EDC_MISS_CLEAN/ EDC_MISS_DIRTY
- likwid-perfctr -C 0-63 -g **HBM_CACHE** ./gpp.knl.ex 512 2 32768 20



```
+----------------------------------------------------+------------+------------+------------+------------+
|                       Metric                       |     Sum    |     Min    |     Max    |     Avg    |
+----------------------------------------------------+------------+------------+------------+------------+
|              Runtime (RDTSC) [s] STAT              |   896.4352 |    14.0068 |    14.0068 |    14.0068 |
|             Runtime unhalted [s] STAT              |   390.2173 |     6.0393 |     9.6183 |     6.0971 |
|                  Clock [MHz] STAT                  | 95979.5220 |  1499.6763 |  1499.6807 |  1499.6800 |
|                      CPI STAT                      |    83.4239 |     1.2985 |     1.5496 |     1.3035 |
|    MCDRAM Memory read bandwidth [MBytes/s] STAT    | 63246.3054 |          0 | 63246.3054 |   988.2235 |
|    MCDRAM Memory read data volume [GBytes] STAT    |   885.8769 |          0 |   885.8769 |    13.8418 |
| MCDRAM Memory writeback bandwidth [MBytes/s] STAT  |   468.4857 |          0 |   468.4857 |     7.3201 |
| MCDRAM Memory writeback data volume [GBytes] STAT  |     6.5620 |          0 |     6.5620 |     0.1025 |
|       MCDRAM Memory bandwidth [MBytes/s] STAT      | 63714.7910 |          0 | 63714.7910 |   995.5436 |
|       MCDRAM Memory data volume [GBytes] STAT      |   892.4389 |          0 |   892.4389 |    13.9444 |
|      DDR Memory read bandwidth [MBytes/s] STAT     |  2569.3065 |          0 |  2569.3065 |    40.1454 |
|      DDR Memory read data volume [GBytes] STAT     |    35.9877 |          0 |    35.9877 |     0.5623 |
|   DDR Memory writeback bandwidth [MBytes/s] STAT   |    21.1772 |          0 |    21.1772 |     0.3309 |
|   DDR Memory writeback data volume [GBytes] STAT   |     0.2966 |          0 |     0.2966 |     0.0046 |
|        DDR Memory bandwidth [MBytes/s] STAT        |  2590.4837 |          0 |  2590.4837 |    40.4763 |
|        DDR Memory data volume [GBytes] STAT        |    36.2843 |          0 |    36.2843 |     0.5669 |
+----------------------------------------------------+------------+------------+------------+------------+
```

L2 GB/s

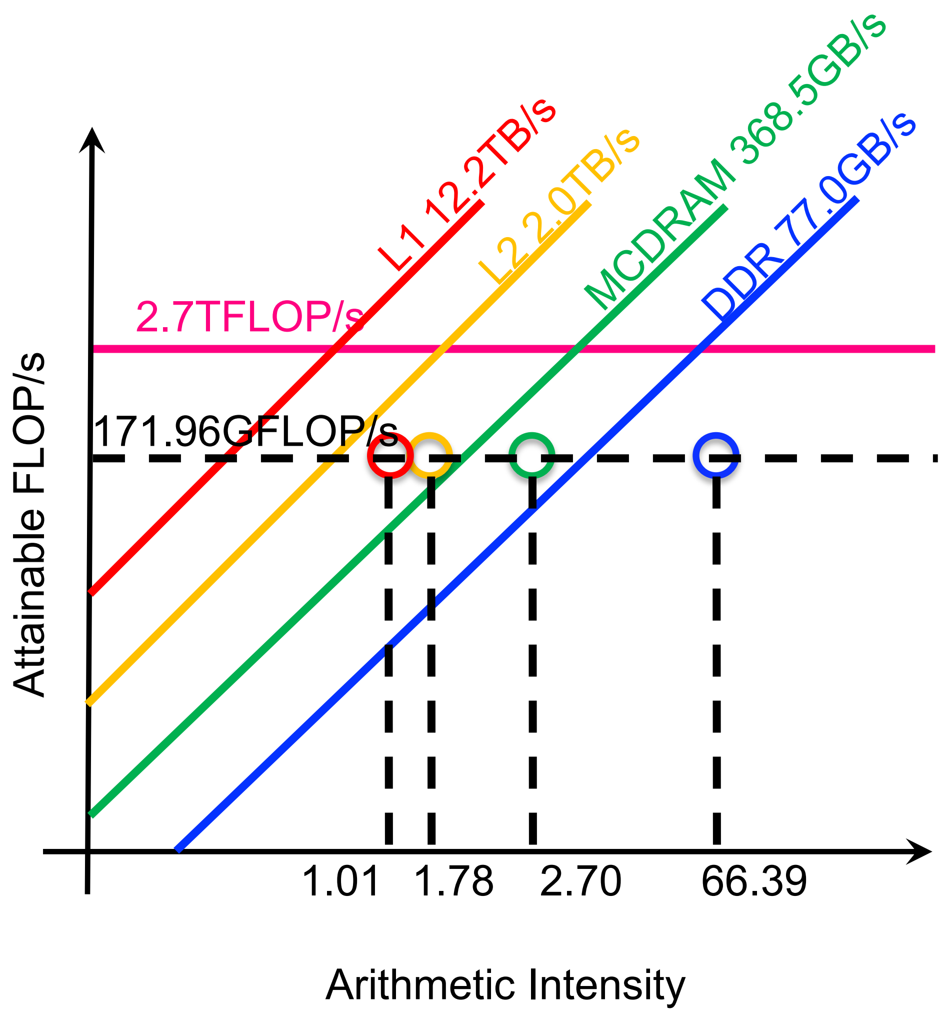
- kernel on KNL: L2 96.80GB/s
  - L2_REQUESTS_REFERENCE
  - OFFCORE_RESPONSE_0_OPTIONS
- likwid-perfctr -C 0-63 -g L2 ./gpp.knl.ex 512 2 32768 20



```
+-------------------------------------+--------------+-------------+-------------+-------------+
|                Metric               |      Sum     |     Min     |     Max     |     Avg     |
+-------------------------------------+--------------+-------------+-------------+-------------+
|        Runtime (RDTSC) [s] STAT     |     895.5200 |     13.9925 |     13.9925 |     13.9925 |
|        Runtime unhalted [s] STAT    |     392.3078 |      6.0719 |      9.6599 |      6.1298 |
|            Clock [MHz] STAT         |   95999.4279 |   1499.9861 |   1499.9914 |   1499.9911 |
|                CPI STAT             |      83.8844 |      1.3055 |      1.5567 |      1.3107 |
| L2 non-RFO bandwidth [MBytes/s] STAT |   96803.9243 |   1498.7686 |   1904.3169 |   1512.5613 |
|  L2 non-RFO data volume [GByte] STAT |    1354.5272 |     20.9715 |     26.6461 |     21.1645 |
|   L2 RFO bandwidth [MBytes/s] STAT   |            0 |           0 |           0 |           0 |
|    L2 RFO data volume [GByte] STAT   |            0 |           0 |           0 |           0 |
|     L2 bandwidth [MBytes/s] STAT     |   96803.9243 |   1498.7686 |   1904.3169 |   1512.5613 |
|      L2 data volume [GByte] STAT     | 1.354528e+06 |  20971.5004 |  26646.1299 |  21164.4950 |
+-------------------------------------+--------------+-------------+-------------+-------------+
```

Resultant Roofline

▪ AI (DRAM):	66.39
▪ AI (MCDRAM):	2.70
▪ AI (L2):	1.78
▪ AI (L1):	1.01
▪ Performance:	171.960 GFLOPS/s



Marking Specific Regions

```
#include <likwid.h>
......
LIKWID_MARKER_INIT;
#pragma omp parallel {
    LIKWID_MARKER_THREADINIT;
}
#pragma omp parallel {
    LIKWID_MARKER_START("foo");
    #pragma omp for
    for(i = 0; i < N; i++) {
            data[i] = omp_get_thread_num();
    }
    LIKWID_MARKER_STOP("foo");
}
LIKWID_MARKER_CLOSE;
```

focus on specific code regions

- `cc -qopenmp -DLIKWID_PERFMON -I$LIKWID_INCLUDE -L$LIKWID_LIB -llikwid -dynamic test.c -o test.x`
- `likwid-perfctr -C 0-3 -g MEM -m ./test.x`

Why isn’t LIKWID good enough?

- LIKWID counts vector uops
- KNL vuop counters aren’t…
  - VL-aware
  - precision-aware
  - mask-aware
  - FMA-aware
- Counters don’t differentiate instruction types (FP, int, shuffle, …)
- **Flop counters were broken on Haswell.**
- Thus, LIKWID might be a good starting point, but its not perfect.

➢ **Need tools that actually count flops correctly and ones that can be used to understand nuances of instruction mixes.**

Intel Software Development Emulator (SDE)

- Dynamic instruction tracing
  - ✓ Accounts for actual loop lengths and branches
  - ✓ Counts instruction types, lengths, etc…
  - ✓ Can mark individual regions
  - ✓ Support for MPI+OpenMP
  - ✓ Can be used to calculate FLOPs (VL-, FMA-, and precision-aware)
  - ✗ Post processing can be expensive.
  - ✗ No insights into cache behavior or DRAM data movement
  - ✗ X86 only

https://software.intel.com/en-us/articles/intel-software-development-emulator

Compiling with SDE at NERSC

- Makefile…

```
MPICC = cc
CFLAGS = -g -O3 -dynamic -qopenmp -restrict -qopt-streaming-stores always  \
         -DSTREAM_ARRAY_SIZE=400000000 -DNTIMES=50 \
         -I$(VTUNE_AMPLIFIER_XE_2018_DIR)/include
LDFLAGS = -L$(VTUNE_AMPLIFIER_XE_2018_DIR)/lib64 -littnotify

stream_mpi.exe: stream_mpi.c Makefile
      $(MPICC) $(CFLAGS) stream_mpi.c -o stream_mpi.exe $(LDFLAGS)

clean:
      rm -f stream_mpi.exe
```

- module load sde
  make

Running with SDE at NERSC

```
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 -- foo.exe
```

- -ivb is used to target Edison's Ivy Bridge ISA (for Cori use -hsw for Haswell or -knl for KNL 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)

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

Parsing the Output

- When the job completes, you'll have a series of files prefixed with "sde_".
- Parse the output to summarize the results…

```
./parse-sde.sh sde_2p16t*
```

- Use the "**Total FLOPs**" line as the numerator in all AI's and performance
- Use the "**Total Bytes**" line as the denominator in the L1 AI
- Can infer vectorization rates and precision

```
$ ./parse-sde.sh sde_2p16t*
Search stanza is "EMIT_GLOBAL_DYNAMIC_STATS"
elements_fp_single_1 = 0
elements_fp_single_2 = 0
elements_fp_single_4 = 0
elements_fp_single_8 = 0
elements_fp_single_16 = 0
elements_fp_double_1 = 2960
elements_fp_double_2 = 0
elements_fp_double_4 = 999999360
elements_fp_double_8 = 0
--->Total single-precision FLOPs = 0
--->Total double-precision FLOPs = 4000000400
--->Total FLOPs = 4000000400
mem-read-1 = 8618384
mem-read-2 = 1232
mem-read-4 = 137276433
mem-read-8 = 149329207
mem-read-16 = 1999998720
mem-read-32 = 0
mem-read-64 = 0
mem-write-1 = 264992
mem-write-2 = 560
mem-write-4 = 285974
mem-write-8 = 14508338
mem-write-16 = 0
mem-write-32 = 499999680
mem-write-64 = 0
--->Total Bytes read = 33752339756
--->Total Bytes written = 16117466472
--->Total Bytes = 49869806228
```

Marking Regions of Interest for SDE

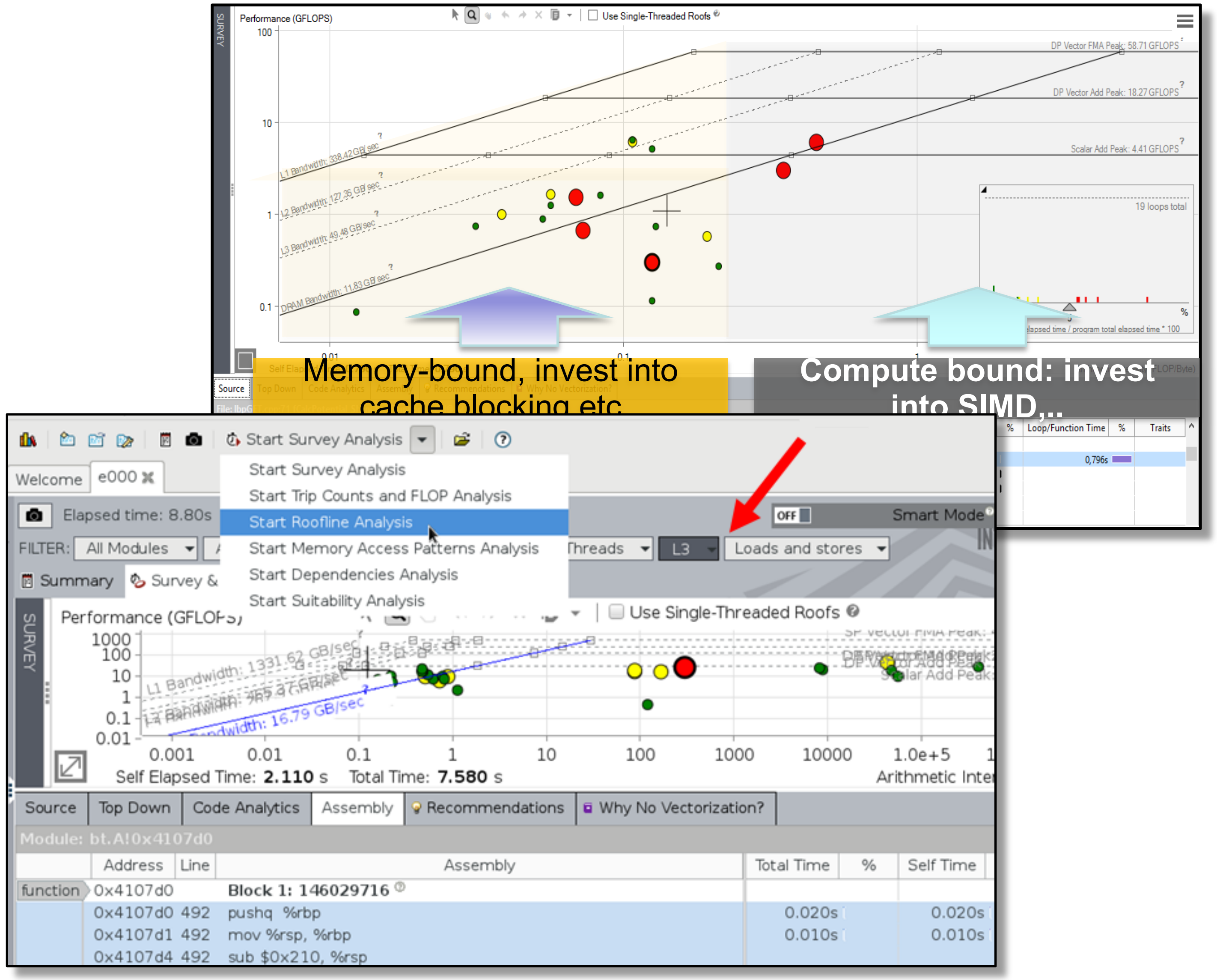
```
// Code must be built with appropriate paths for VTune include file (ittnotify.h) and
   library (-littnotify)
   #include <ittnotify.h>

   __SSC_MARK(0x111); // start SDE tracing, note it uses 2 underscores
   __itt_resume();    // start VTune, again use 2 underscores

   for (k=0; k<NTIMES; k++) {
   #pragma omp parallel for
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
   a[j] = b[j]+scalar*c[j];
   }

   __itt_pause();     // stop VTune
   __SSC_MARK(0x222); // stop SDE tracing
```

Essential when analyzing Individual kernels

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

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


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
Memory-bound, invest into cache blocking etc
Compute bound: invest into SIMD,...
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
FILTER: All Modules
Threads
L3
Loads and stores
OFF
Smart Mode
Summary
Survey &
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
L1 Bandwidth: 1331.62 GB/sec
L3 Bandwidth: 465.47 GB/sec
Bandwidth: 16.79 GB/sec
SP Vector FMA Peak
DP Vector Add Peak
Scalar Add Peak
1000
100
10
1
0.1
0.01
0.001
0.01
0.1
1
10
100
1000
10000
1.0e+5
Arithmetic Inte
Self Elapsed Time: 2.110 s
Total Time: 7.580 s
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
Module: bt.A!0x4107d0

```
function  Address   Line  Assembly                    Total Time  %  Self Time
          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

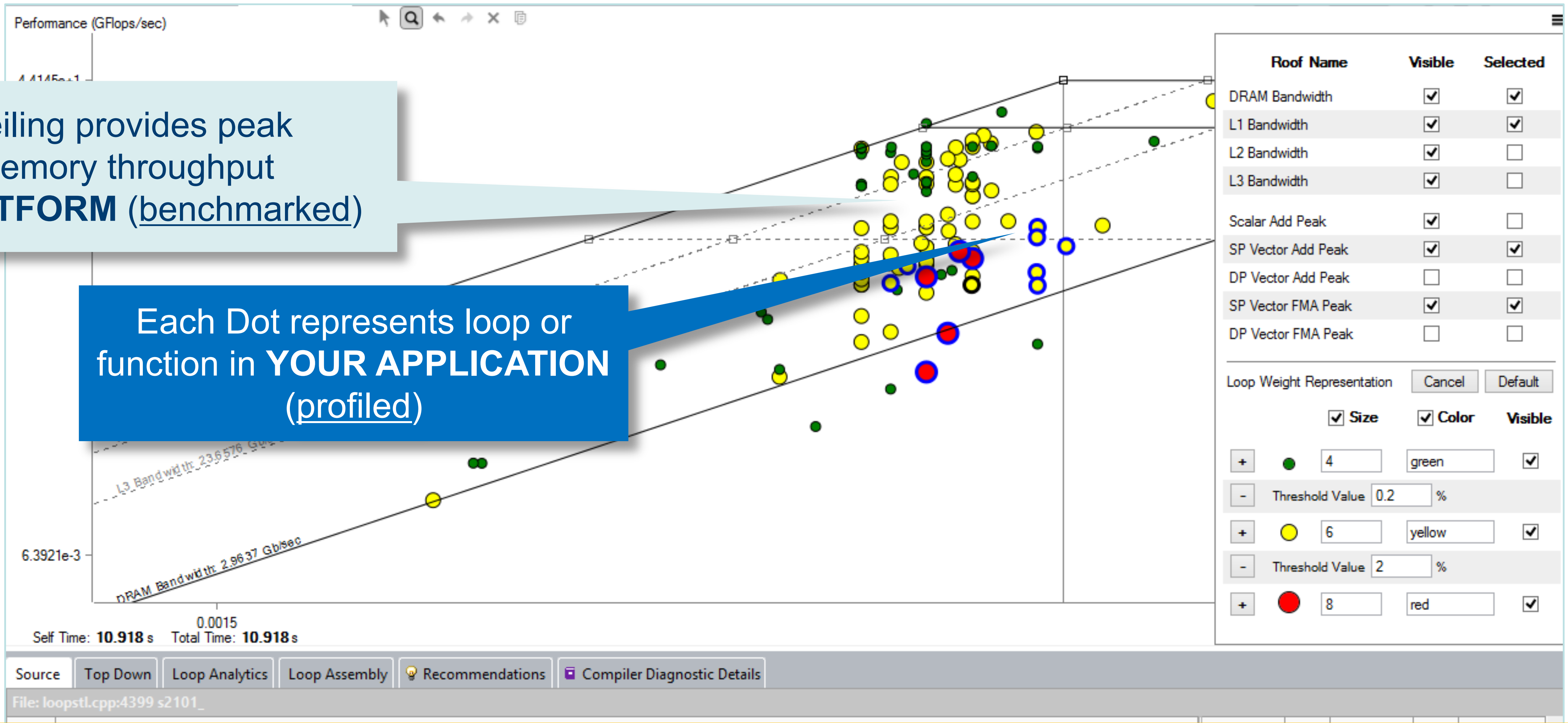
BERKELEY LAB

Intel® Advisor: 2-pass Approach

Roofline: X-Axis (AI): #FLOPs / #Bytes Y-Axis (FLOP/s): #FLOP(mask-aware)/time	**Overhead**
Step 1: Survey (`-collect survey`) • Records run times • User-mode sampling; non-intrusive • ***No need for root access***	**1x**
Step 2: FLOPs (`-collect tripcounts -flops`) • Record #FLOPs, #Bytes, AVX512 masks • Precise, instrumentation-based count of the number of instructions • ***No need for root access***	**3-5x** **(8-37x)[1]**

[1]With Integrated Roofline (Cache Simulator) enabled.

Intel® Advisor: Roofline Automation



Self Time: **10.918** s Total Time: **10.918** s

Roof Name	Visible	Selected
DRAM Bandwidth	✓	✓
L1 Bandwidth	✓	✓
L2 Bandwidth	✓	
L3 Bandwidth	✓	
Scalar Add Peak	✓	
SP Vector Add Peak	✓	✓
DP Vector Add Peak		
SP Vector FMA Peak	✓	✓
DP Vector FMA Peak		

Loop Weight Representation Cancel Default

		✓ Size	✓ Color	Visible
+	●	4	green	✓
-	Threshold Value	0.2	%	
+	●	6	yellow	✓
-	Threshold Value	2	%	
+	●	8	red	✓

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

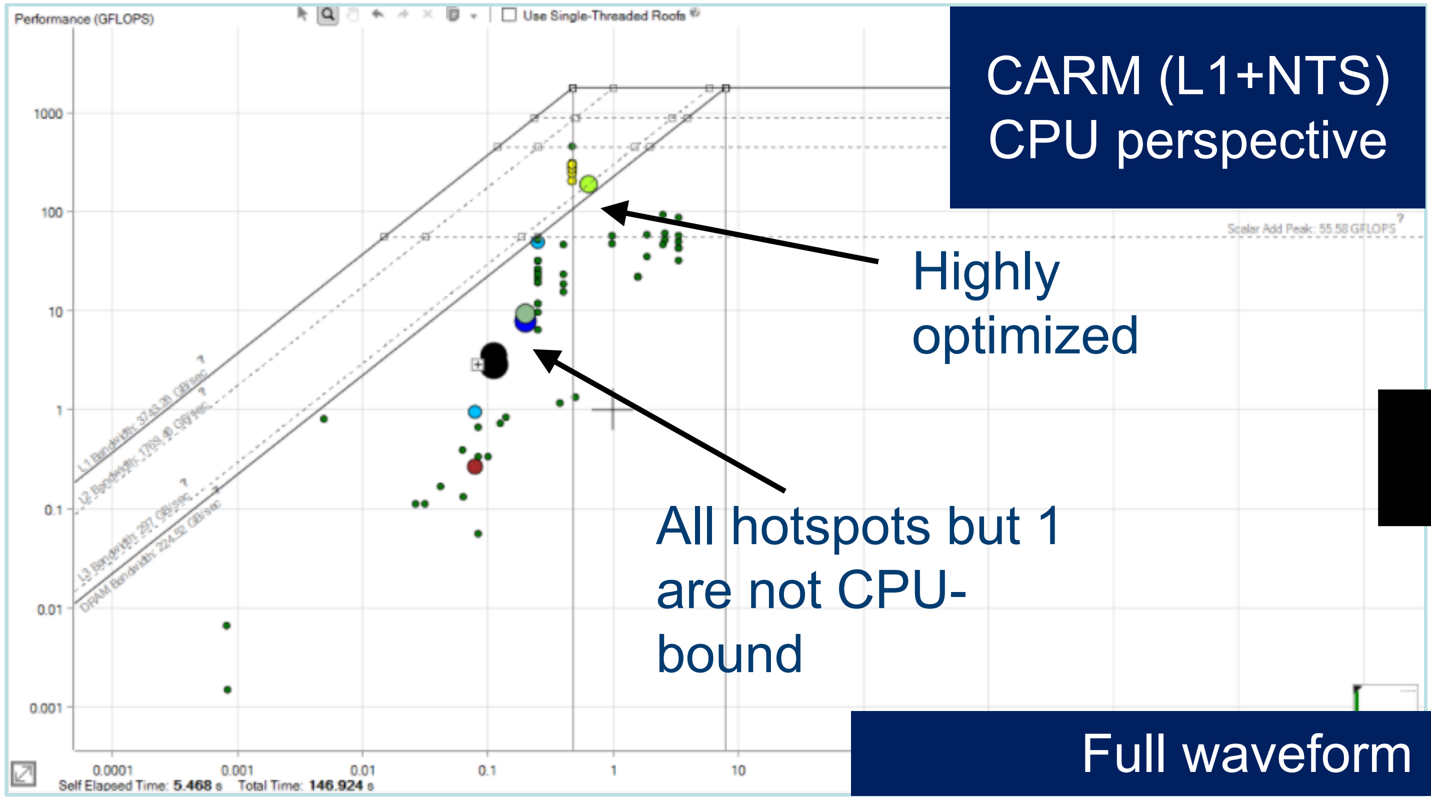
File: loopstl.cpp:4399 s2101_

Each Ceiling provides peak CPU/Memory throughput of your **PLATFORM** (benchmarked)

Each Dot represents loop or function in **YOUR APPLICATION** (profiled)

Automatic and integrated – first class citizen in Intel® Advisor

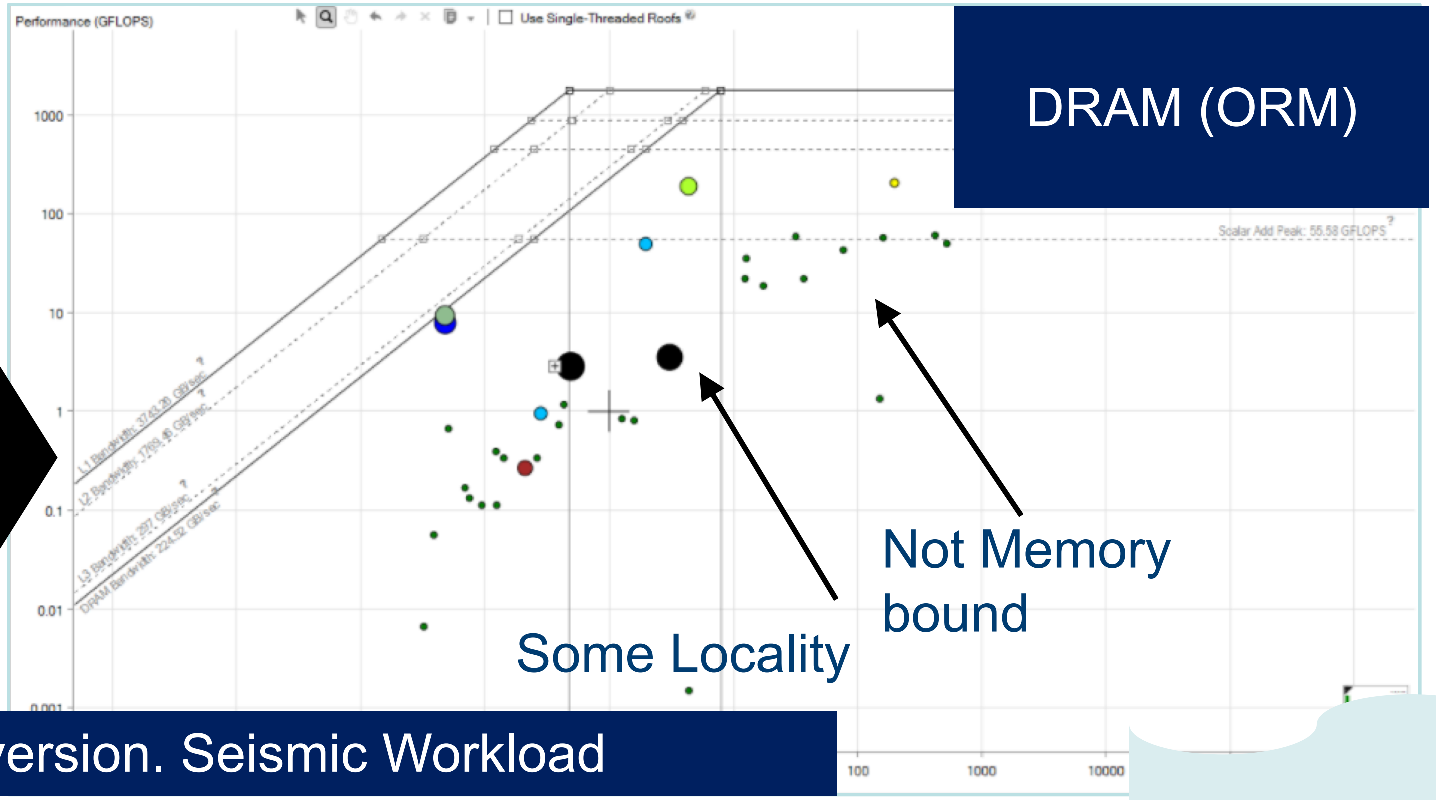
NEW: Integrated Roofline



CARM (L1+NTS) CPU perspective

Highly optimized

All hotspots but 1 are not CPU-bound



DRAM (ORM)

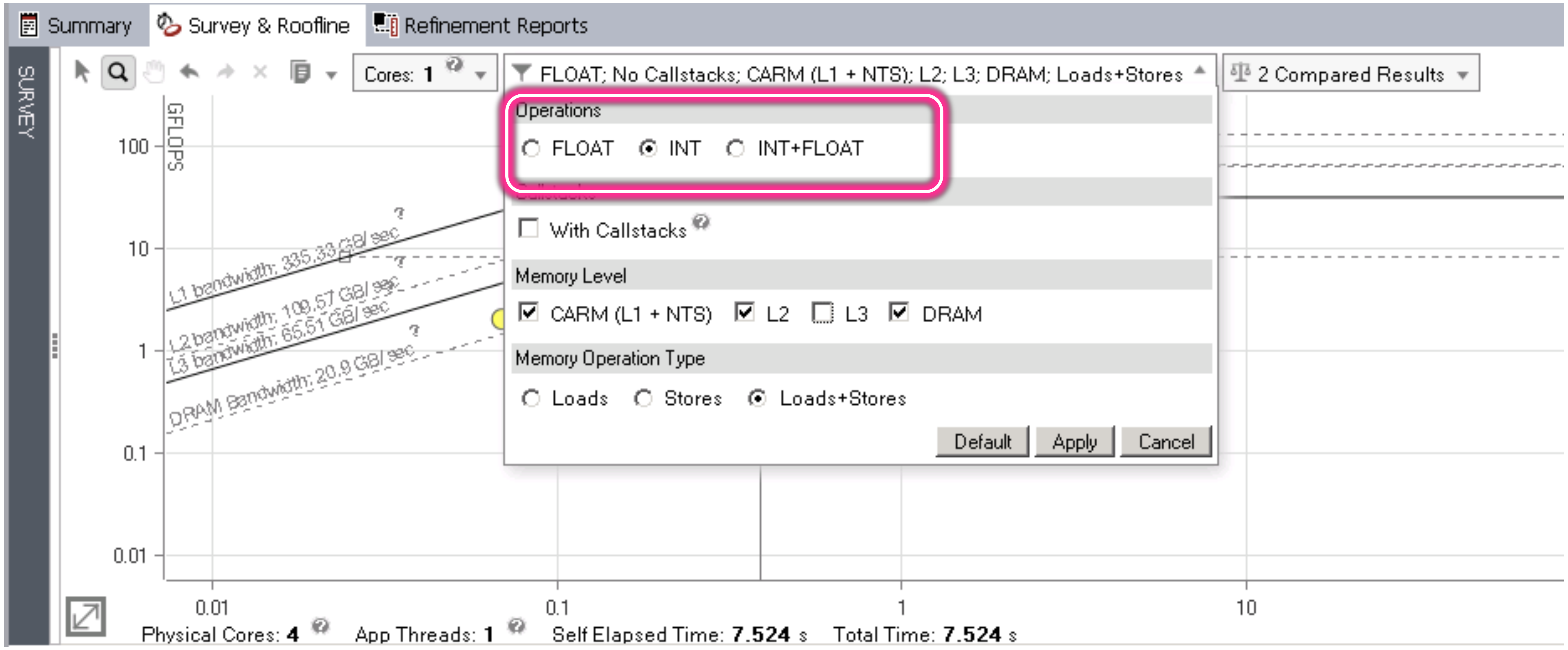
Some Locality

Not Memory bound

Full waveform Inversion. Seismic Workload

Data: Courtesy Philippe Thierry

NEW: Integer, Float, Int+Float Rooflines



Summary | Survey & Roofline | Refinement Reports

Cores: 1

FLOAT; No Callstacks; CARM (L1 + NTS); L2; L3; DRAM; Loads+Stores

2 Compared Results

Operations

FLOAT [✓ INT] INT+FLOAT

Callstacks

With Callstacks ☐

Memory Level

CARM (L1 + NTS) ☑ L2 ☑ L3 ☐ DRAM ☑

Memory Operation Type

Loads Stores [✓ Loads+Stores]

Default | Apply | Cancel

SURVEY
GFLOPS
100
10
1
0.1
0.01
L1 bandwidth: 335.33 GB/ sec
L2 bandwidth: 109.57 GB/ sec
L3 bandwidth: 65.51 GB/ sec
DRAM Bandwidth: 20.9 GB/ sec
0.01
0.1
1
10

Physical Cores: 4 App Threads: 1 Self Elapsed Time: 7.524 s Total Time: 7.524 s

Integrated Roofline Model

Old Approach…

```
source advixe-vars.sh
advixe-cl -collect survey --project-dir ./your_project -- <your-executable-with-parameters>
advixe-cl -collect tripcounts  -enable-cache-simulation -flop --project-dir ./your_project -- <your-
    executable-with-parameters>
```

New Approach (but not compatible with MPI)…

```
source advixe-vars.sh
advixe-cl -collect roofline  -enable-cache-simulation --project-dir ./your_project -- <your-
    executable-with-parameters>
```

(optional) copy data to your UI desktop system

```
advixe-gui ./your_project
```

https://software.intel.com/en-us/articles/integrated-roofline-model-with-intel-advisor

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
```

Tools for Roofline Analysis on GPUs

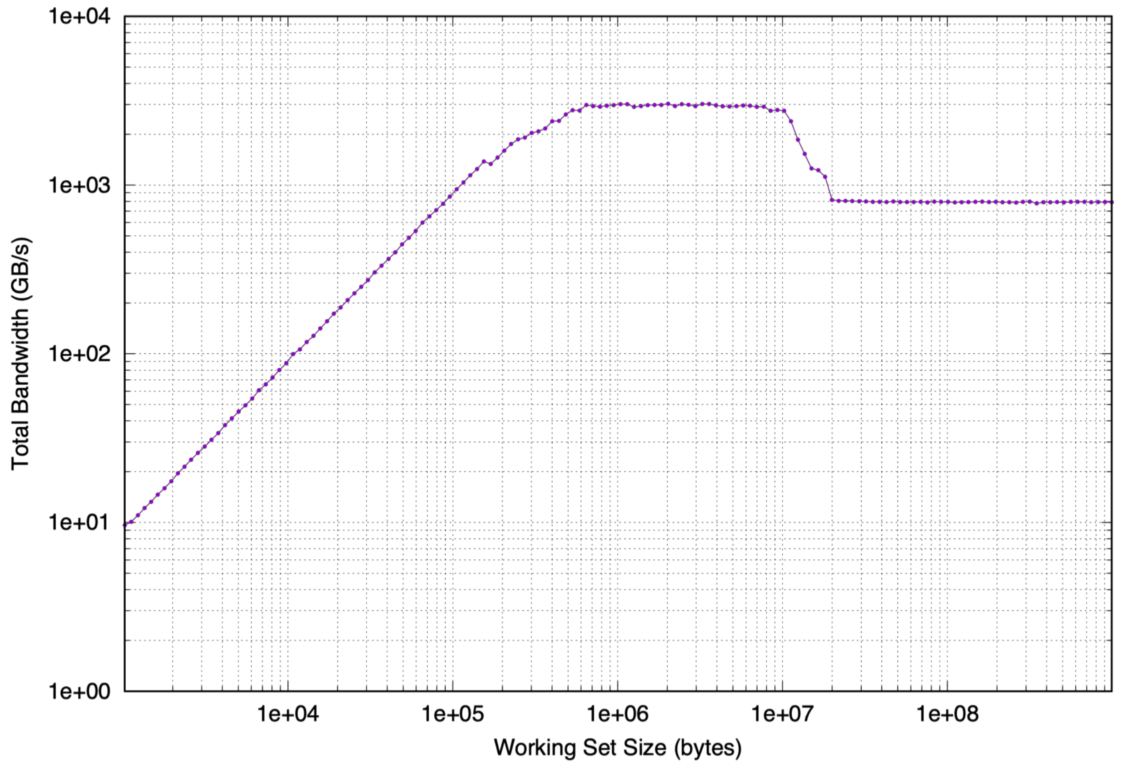
slides provided by Charlene Yang (CJYang@lbl.gov)

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

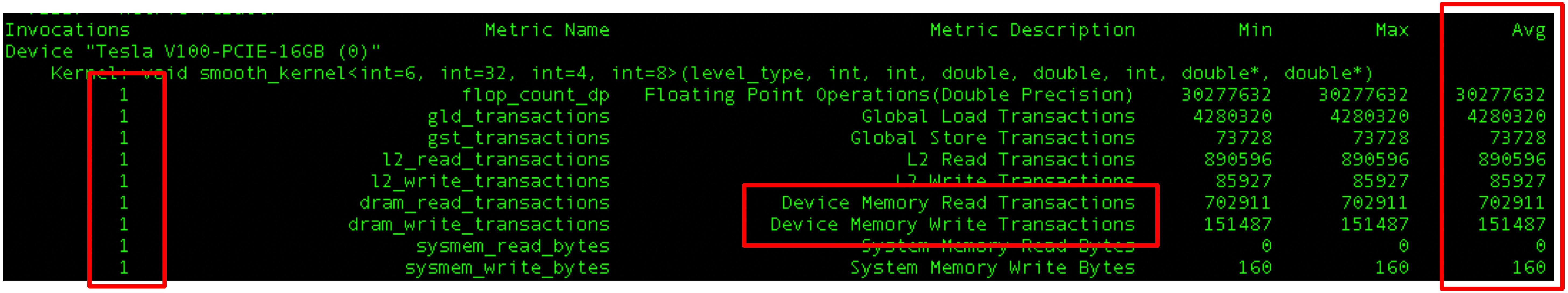
- FLOPs:
  - predication aware, and divides aware, **dp/dp_add/dp_mul/dp_fma**, **sp***
    nvprof --kernels 'kernel_name' --metrics 'flop_count_xx' ./application
- Bytes for different cache levels to construct hierarchical Roofline
  nvprof --kernels 'kernel_name' --metrics 'metric_name'./application
  - 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

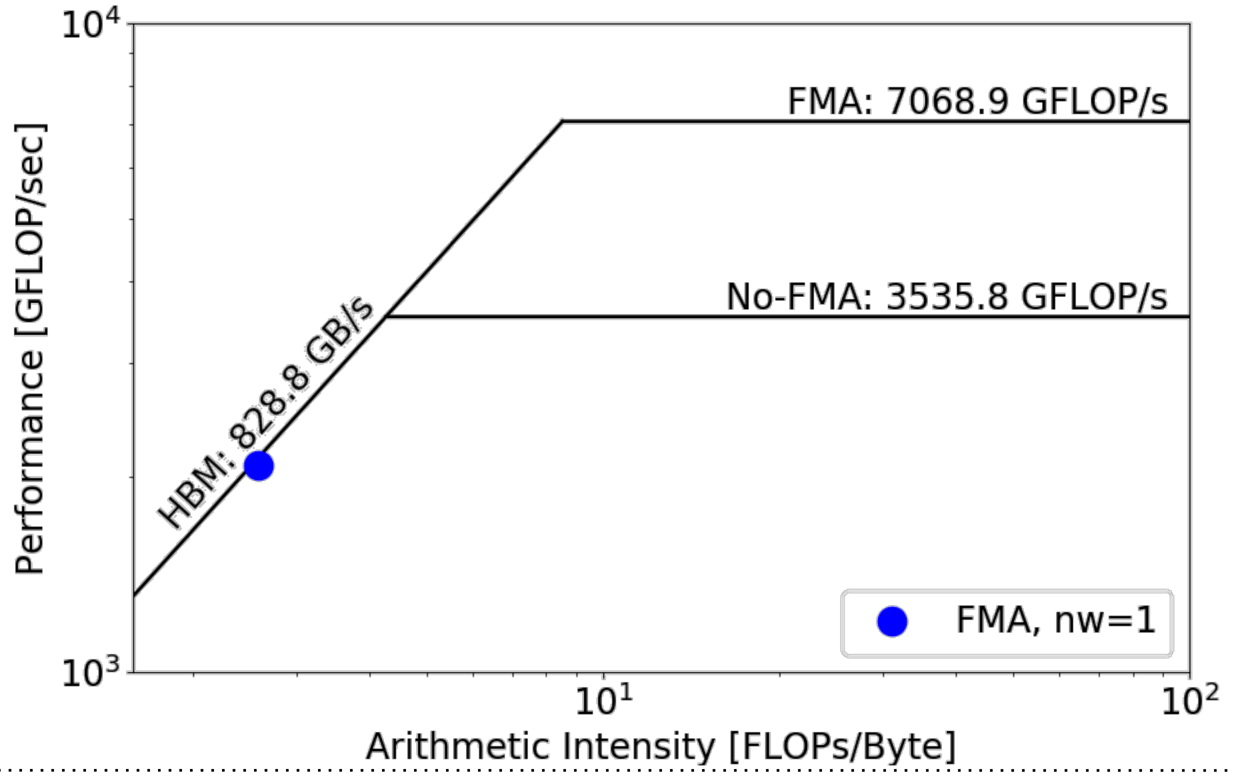


```
Invocations                               Metric Name                            Metric Description         Min         Max         Avg
Device "Tesla V100-PCIE-16GB (0)"
    Kernel: void smooth_kernel<int=6, int=32, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)
          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

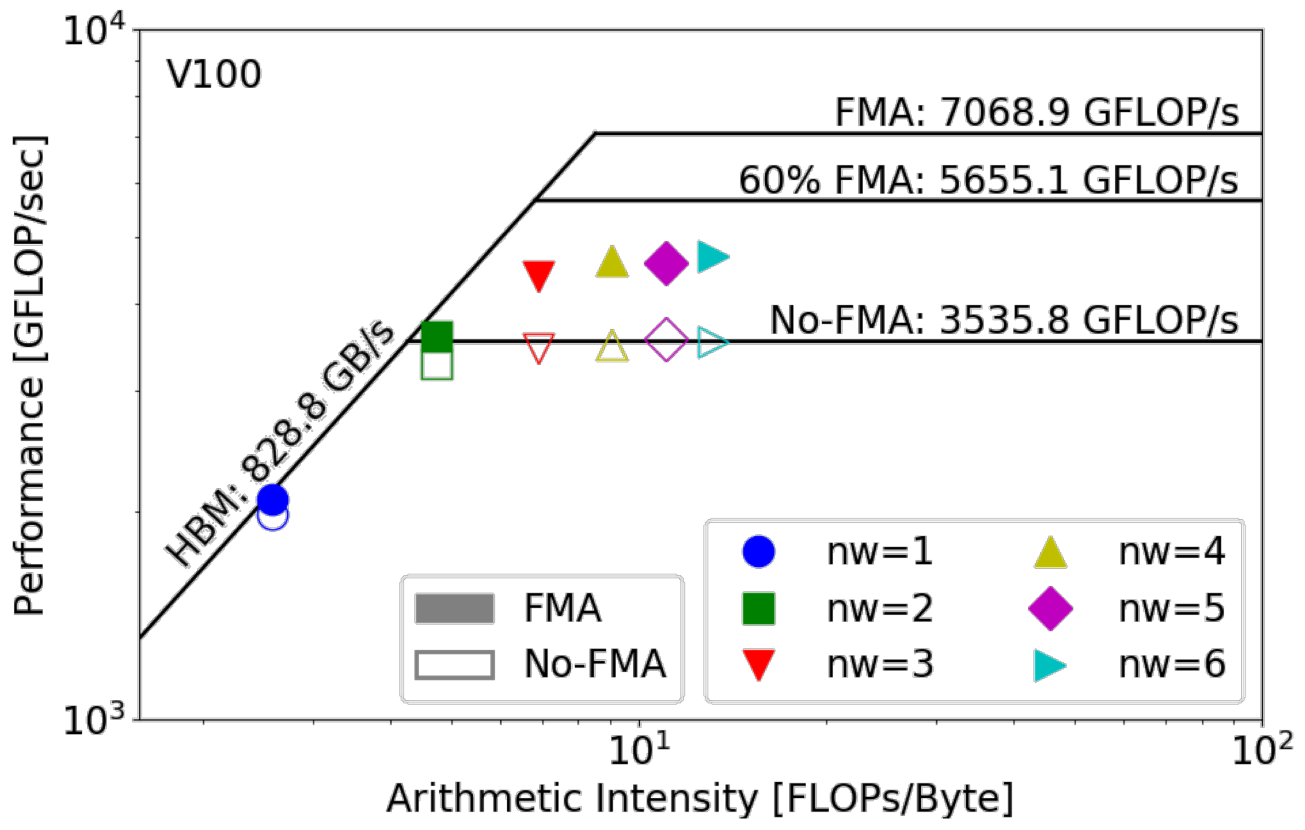
- Python scripts using Matplotlib
  https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting
- Simple example: **`plot_roofline.py data.txt`**
- Tweaking needed for more sophisticated plotting, see examples

```
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

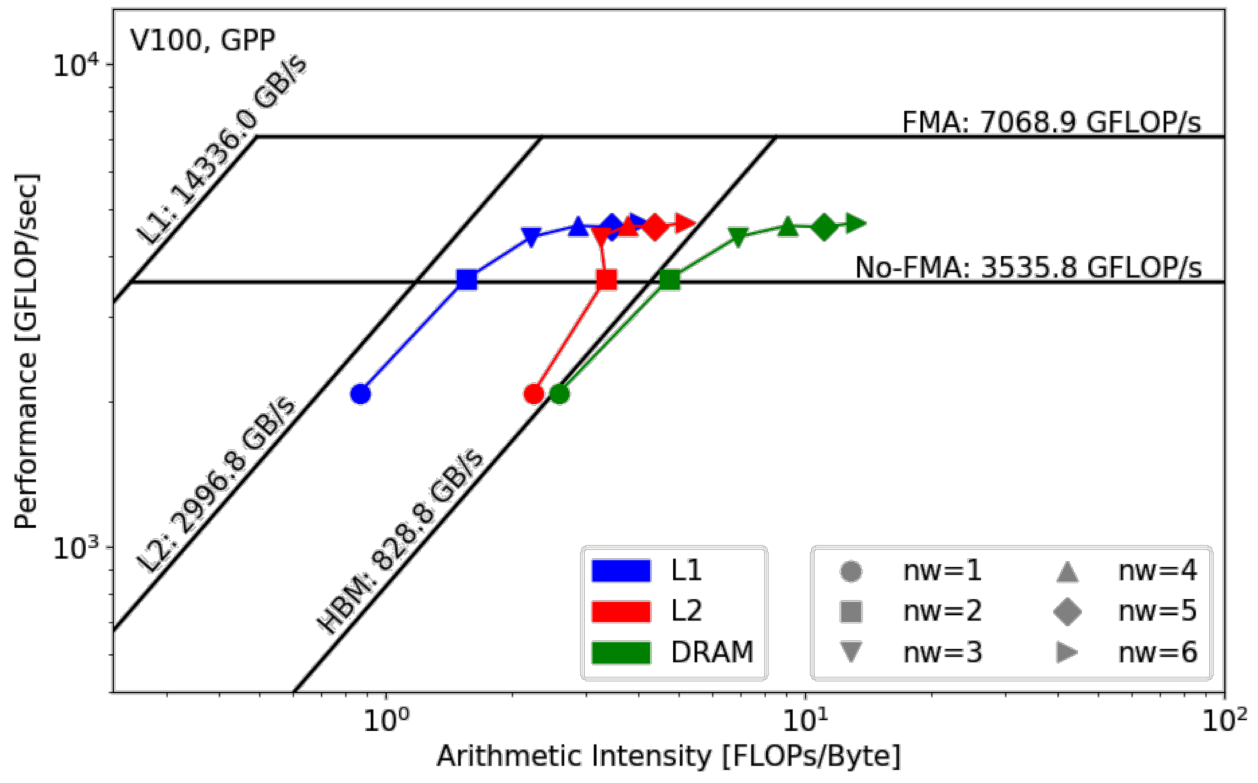
- Use BerkeleyGW Proxy app GPP to see GPU effects
- HBM Roofline
- AI increases as **`nw`** grows
- bandwidth bound → compute bound
- Disable FMA in the compiler…
  - (**`-fmad=true/false`**)
  - "No-FMA" converges to its ceiling
  - But FMA doesn't



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)



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

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

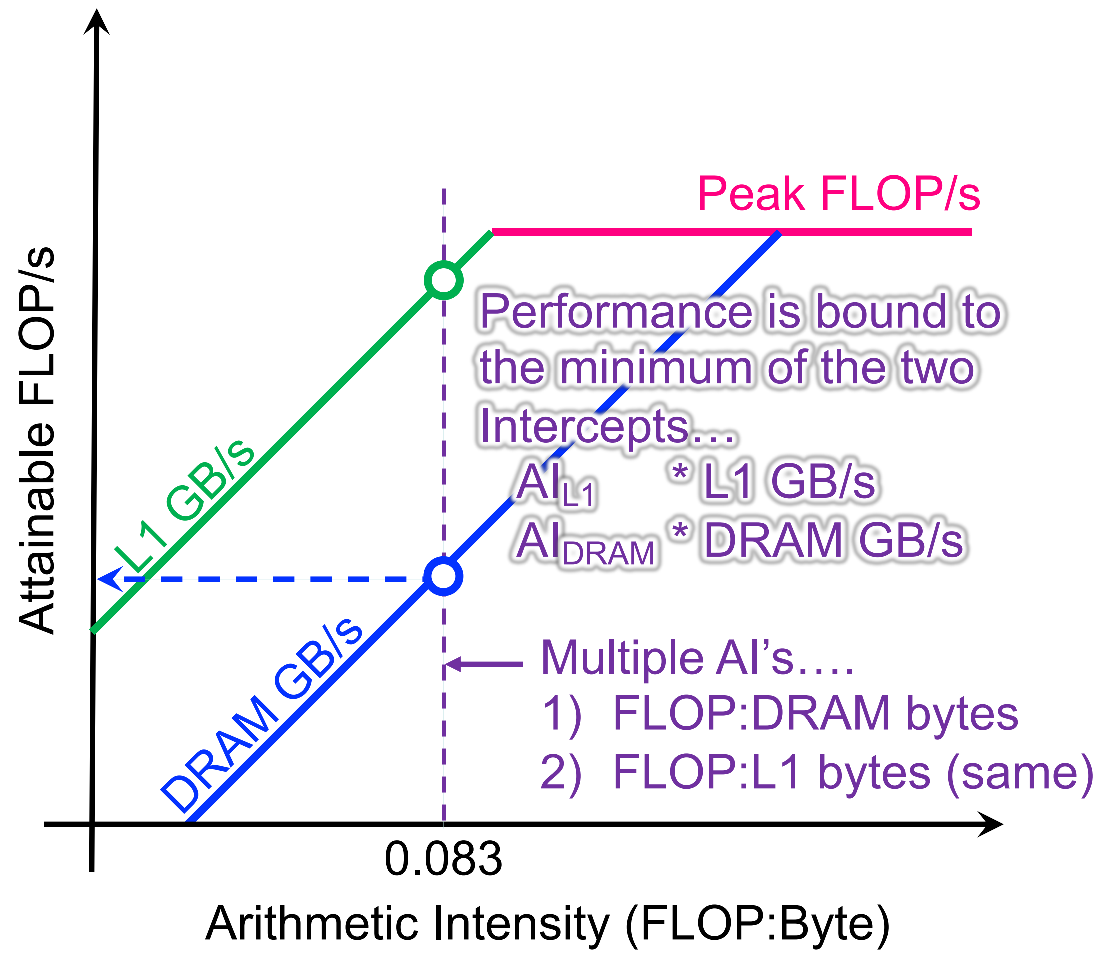
Example: STREAM

- 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

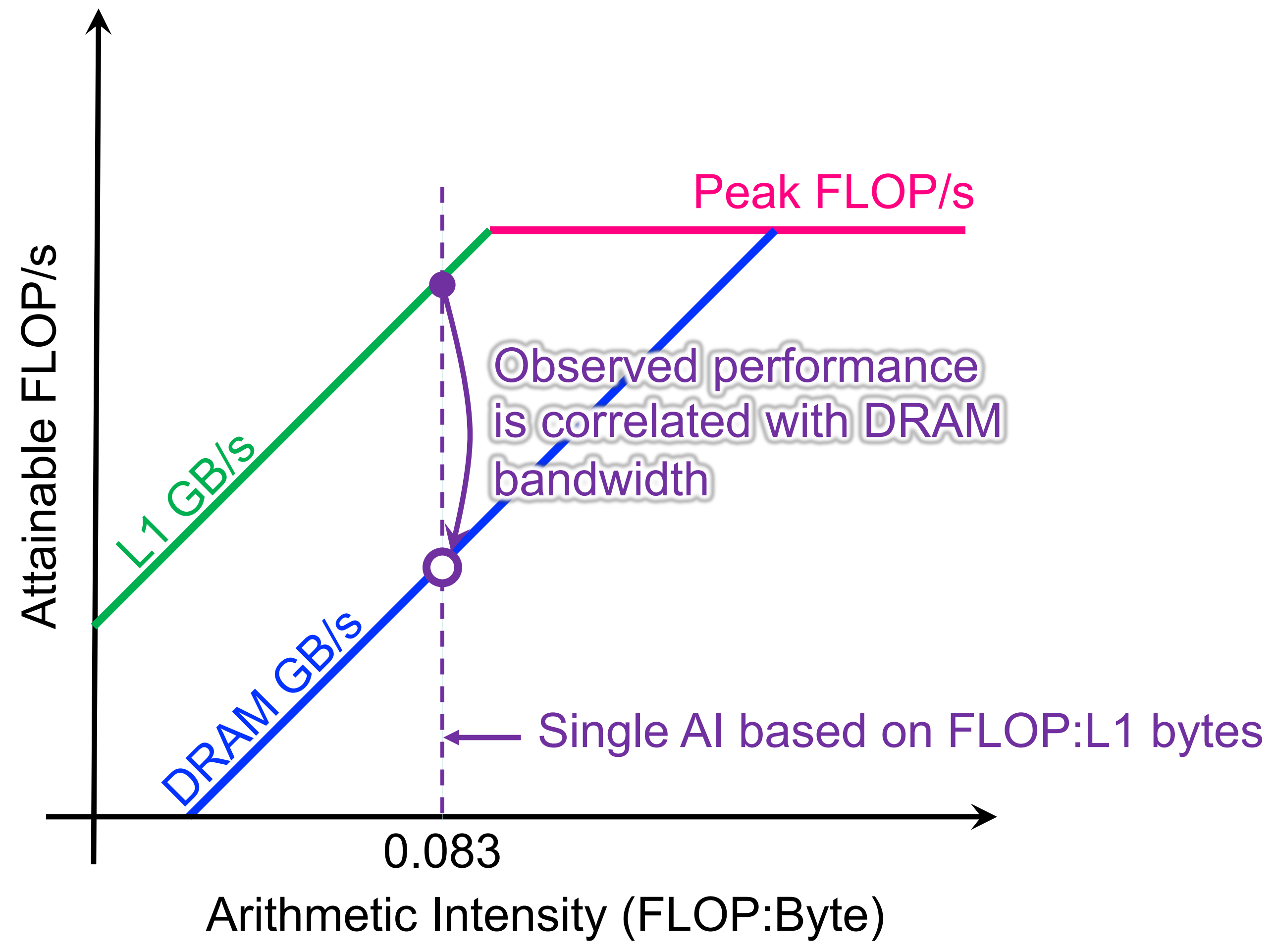
```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



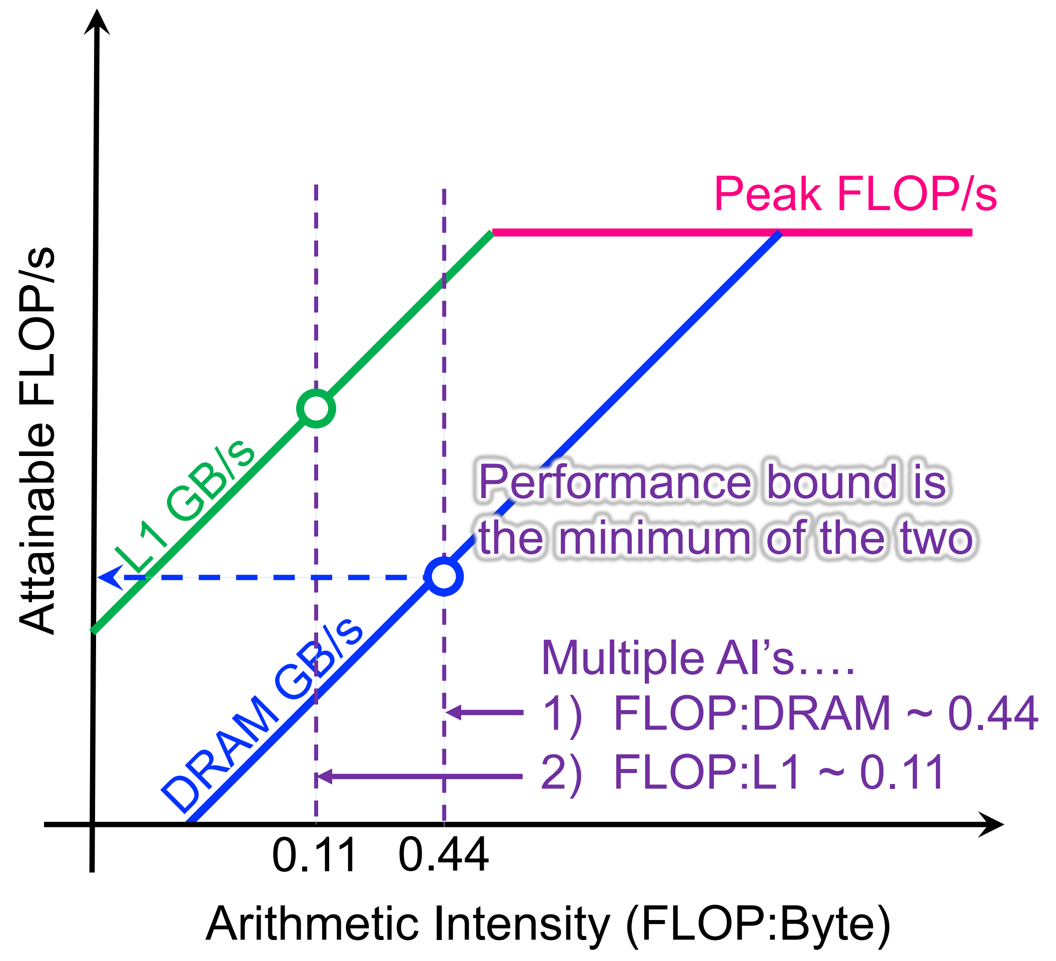
Example: 7-point Stencil (Small Problem)

- 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

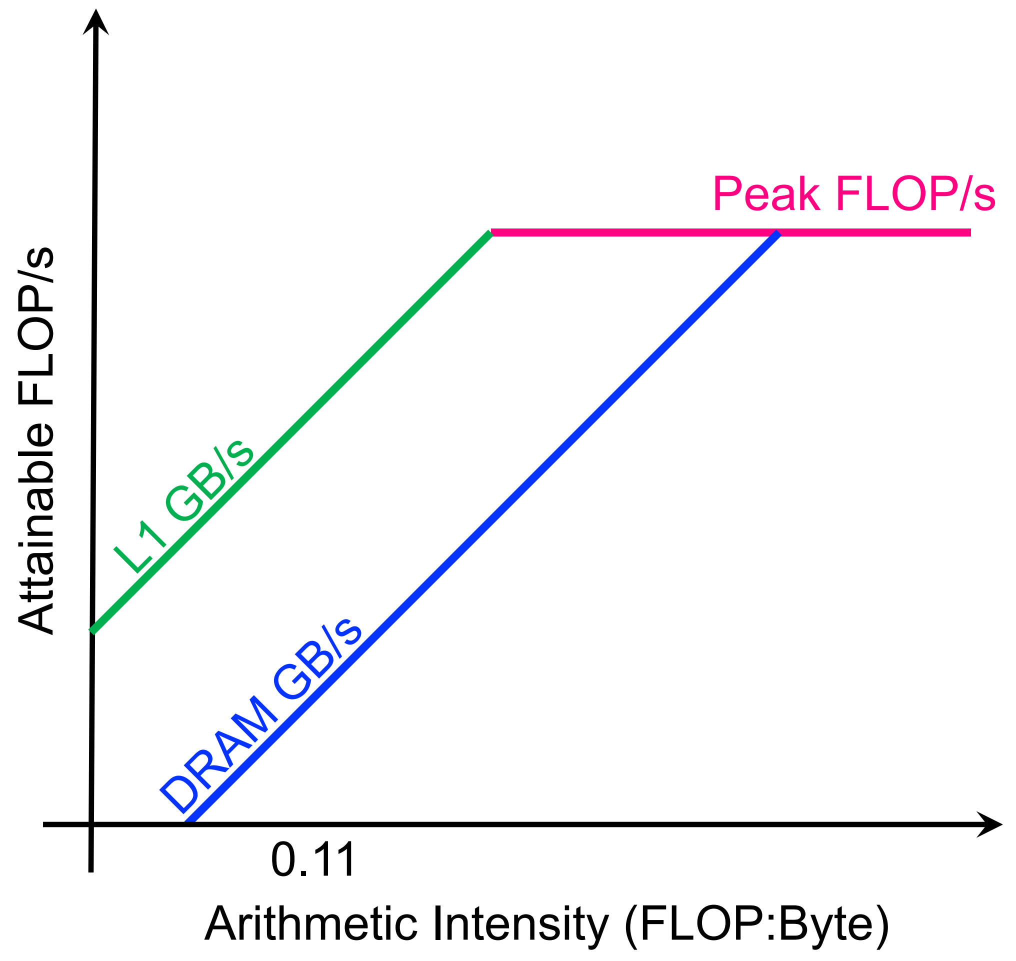
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk        ]
                + old[ijk-1      ]
                + old[ijk+1      ]
                + old[ijk-jStride]
                + old[ijk+jStride]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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

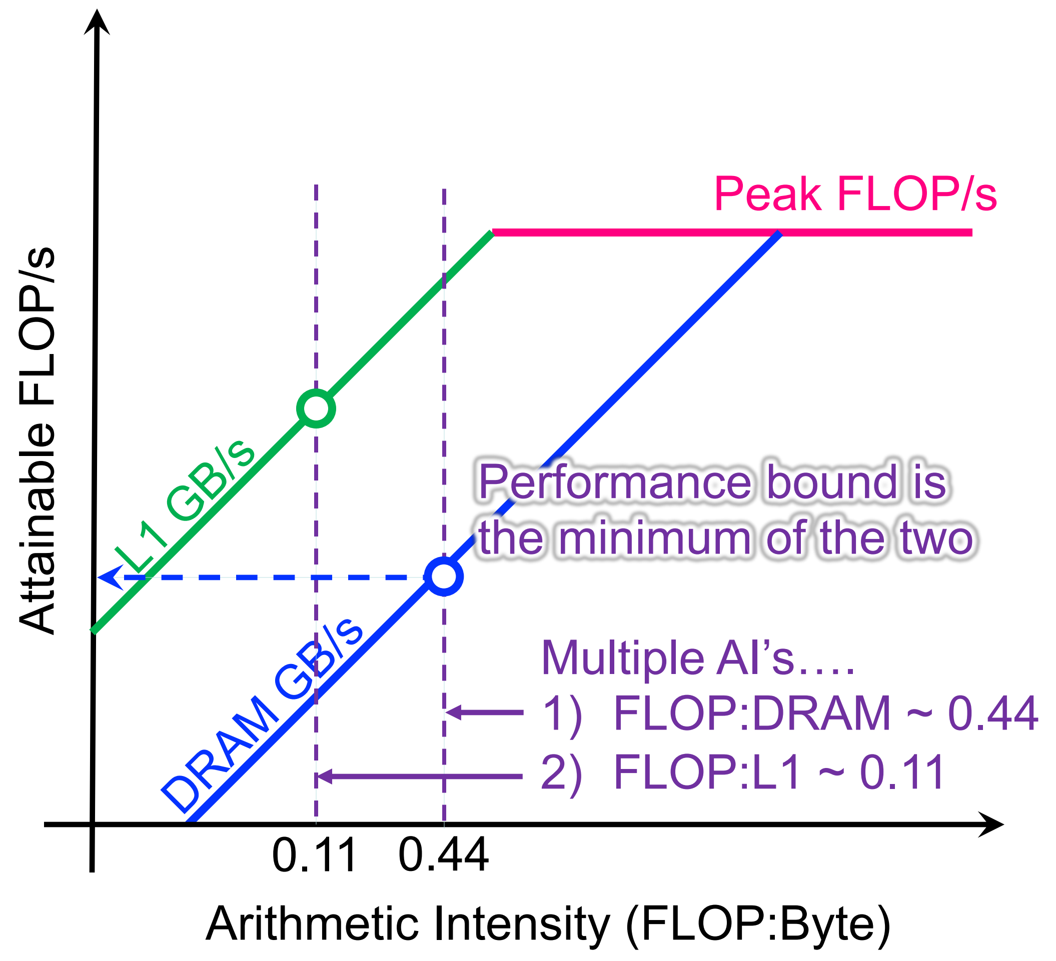


Cache-Aware Roofline

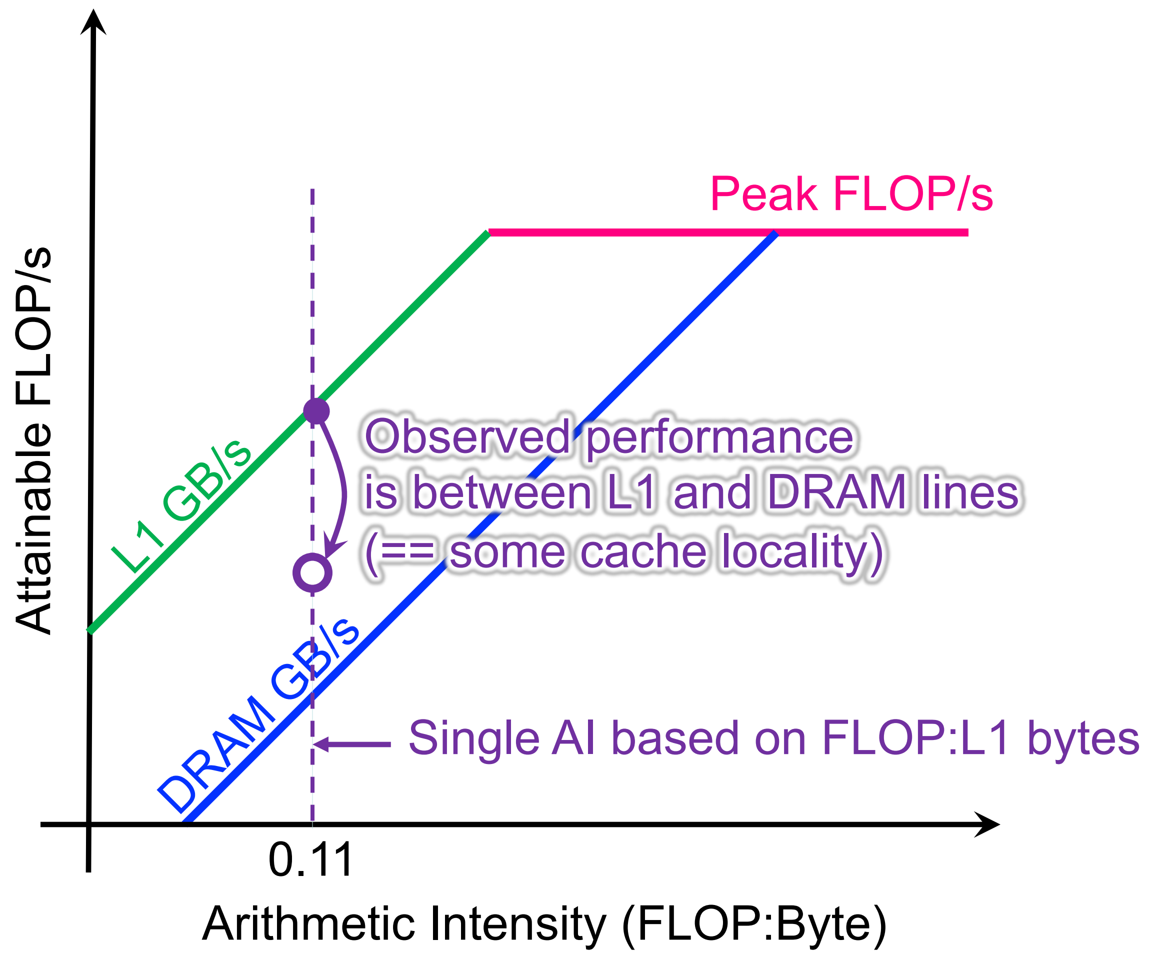


Example: 7-point Stencil (Small Problem)

Hierarchical Roofline



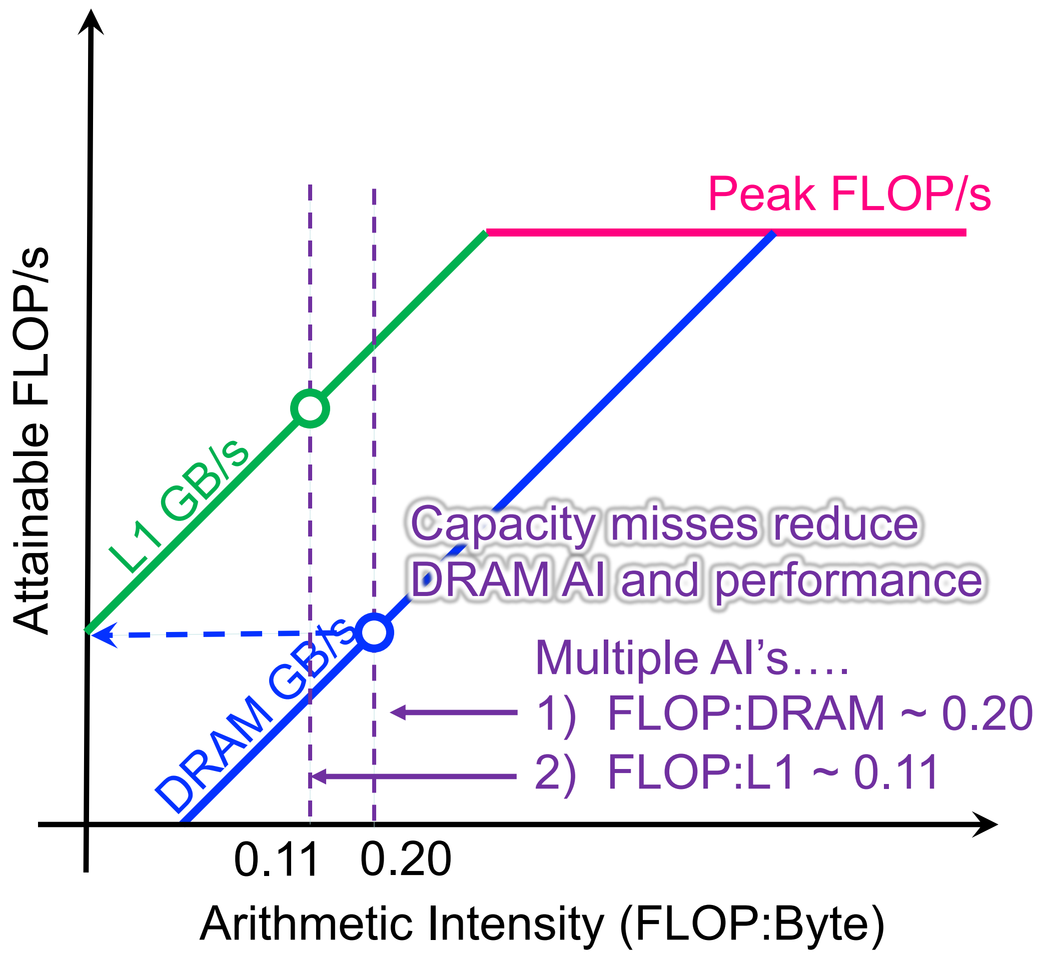
Cache-Aware Roofline



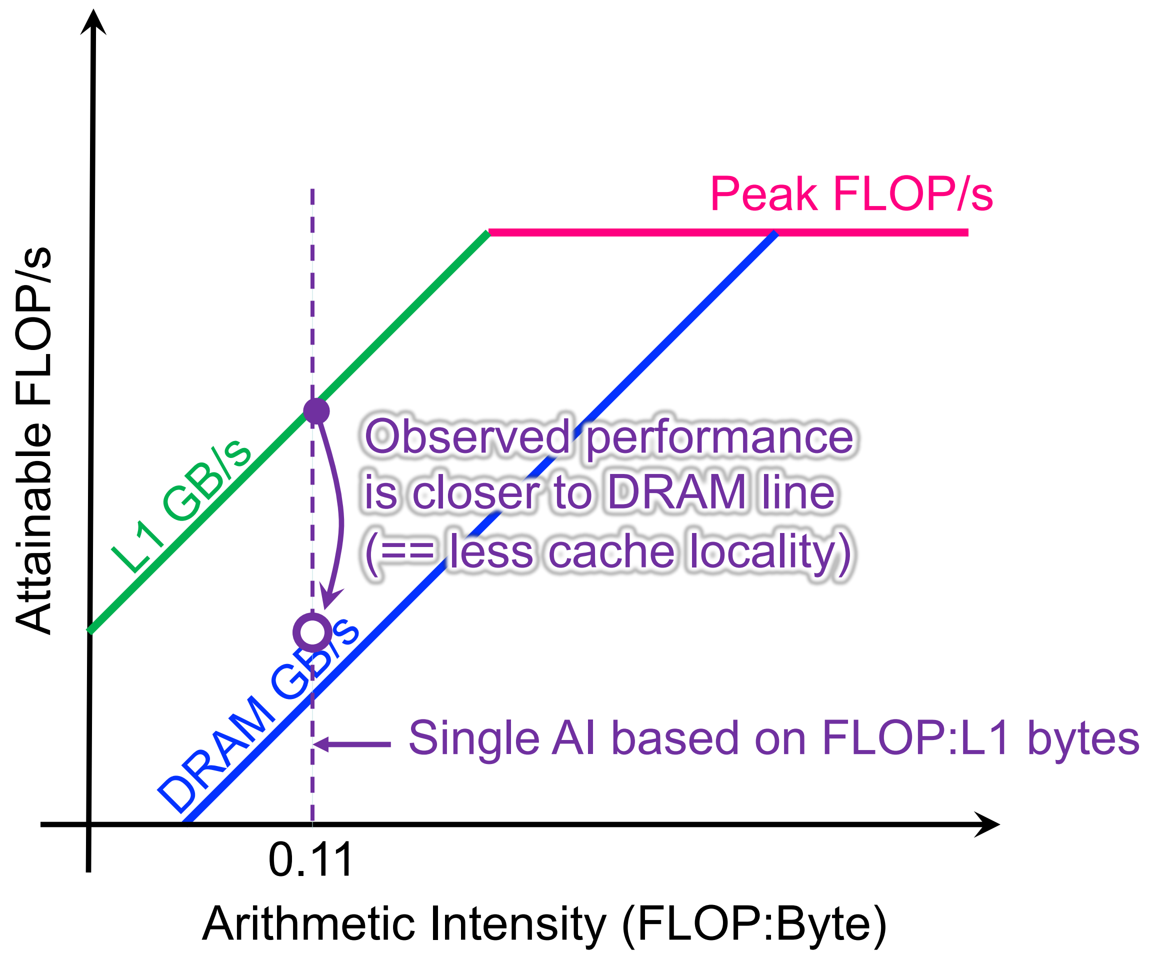
Peak FLOP/s
Attainable FLOP/s
L1 GB/s
DRAM GB/s
Observed performance is between L1 and DRAM lines (== some cache locality)
Single AI based on FLOP:L1 bytes
0.11
Arithmetic Intensity (FLOP:Byte)

Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

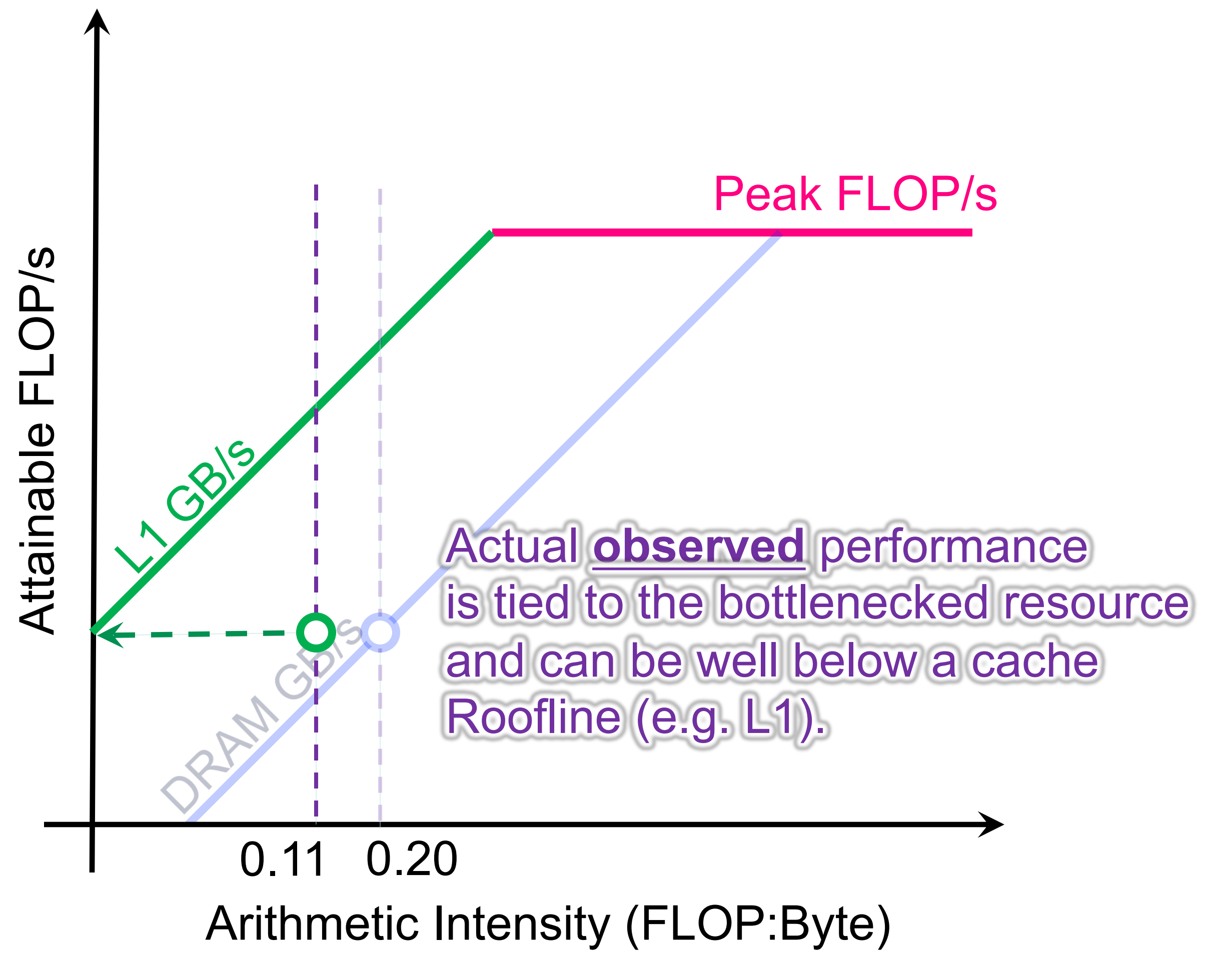


Cache-Aware Roofline

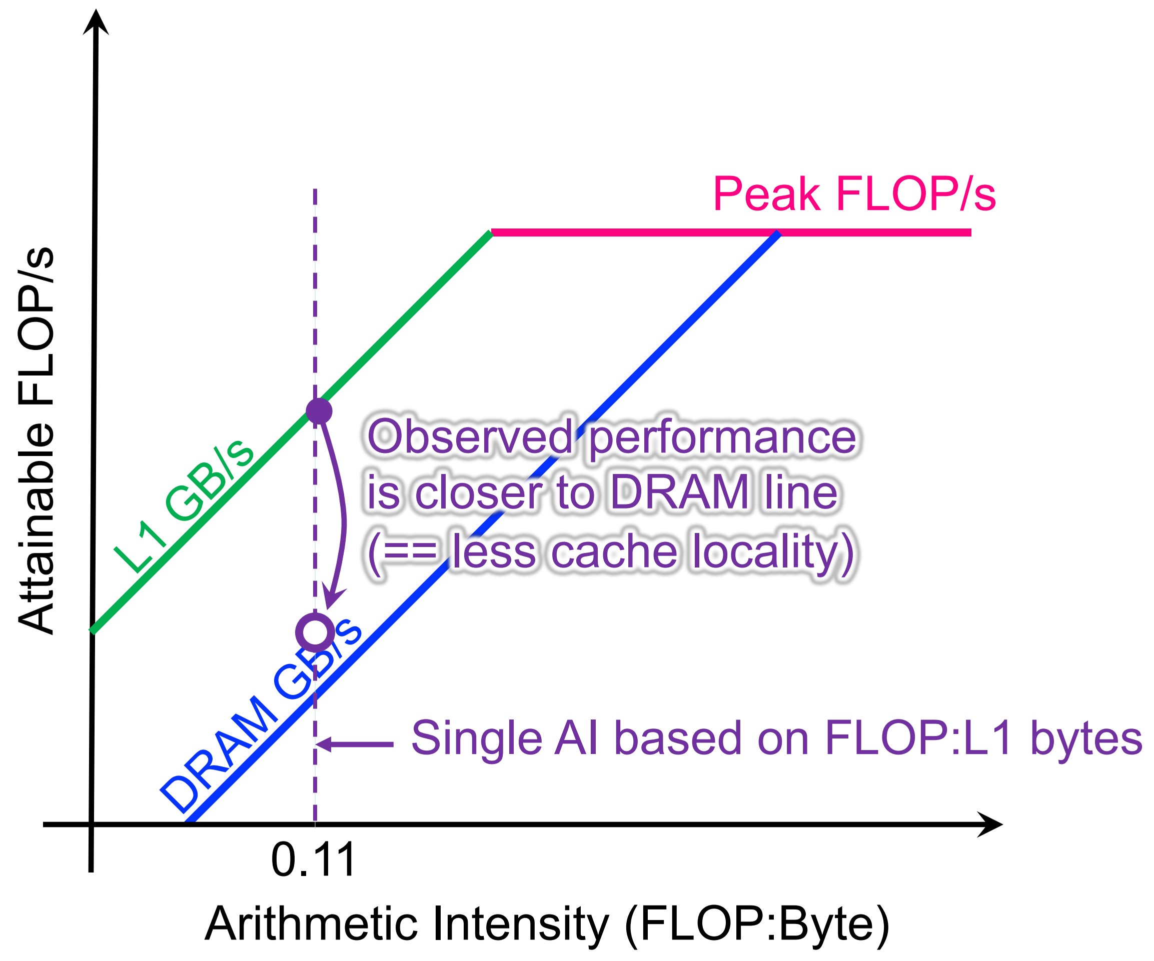


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

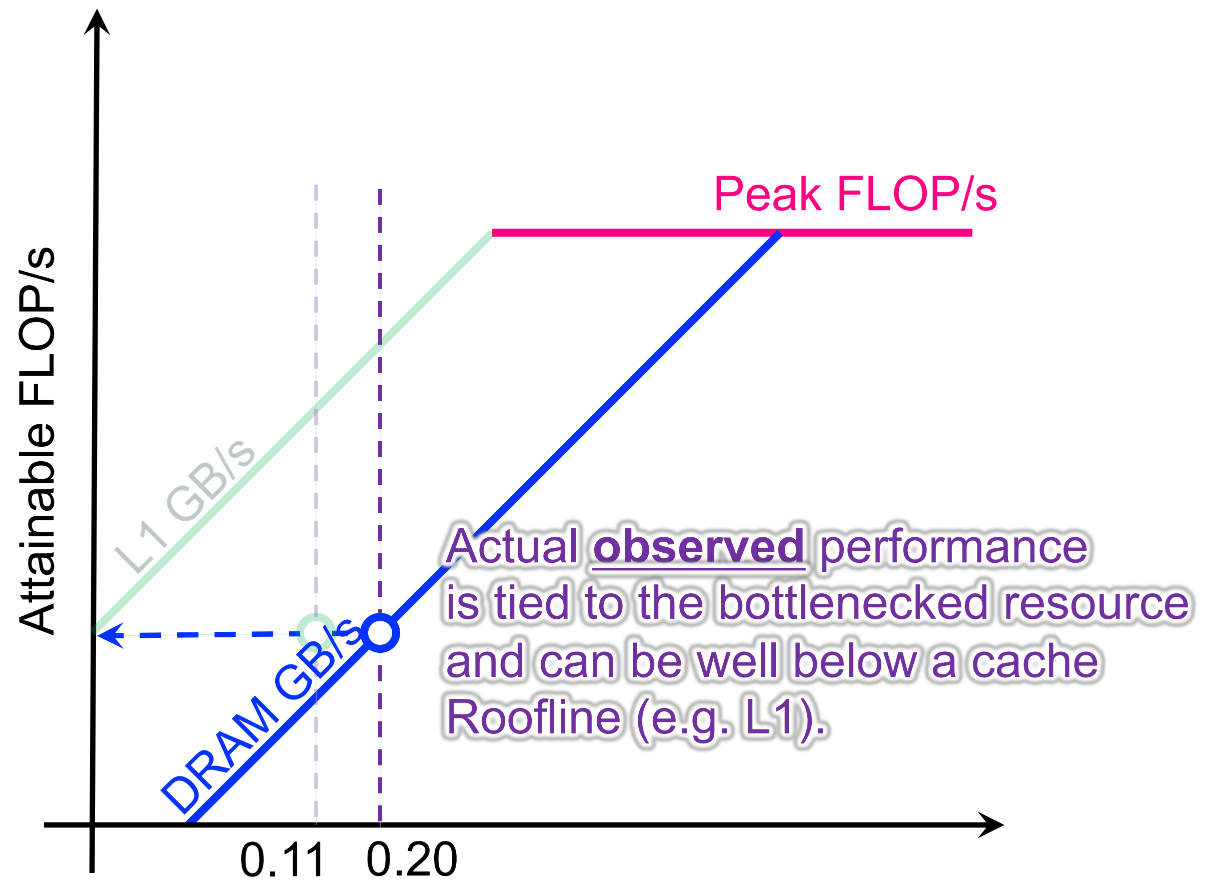

Attainable FLOP/s
Peak FLOP/s
L1 GB/s
DRAM GB/s
Actual **observed** performance
is tied to the bottlenecked resource
and can be well below a cache
Roofline (e.g. L1).
0.11
0.20
Arithmetic Intensity (FLOP:Byte)

Cache-Aware Roofline



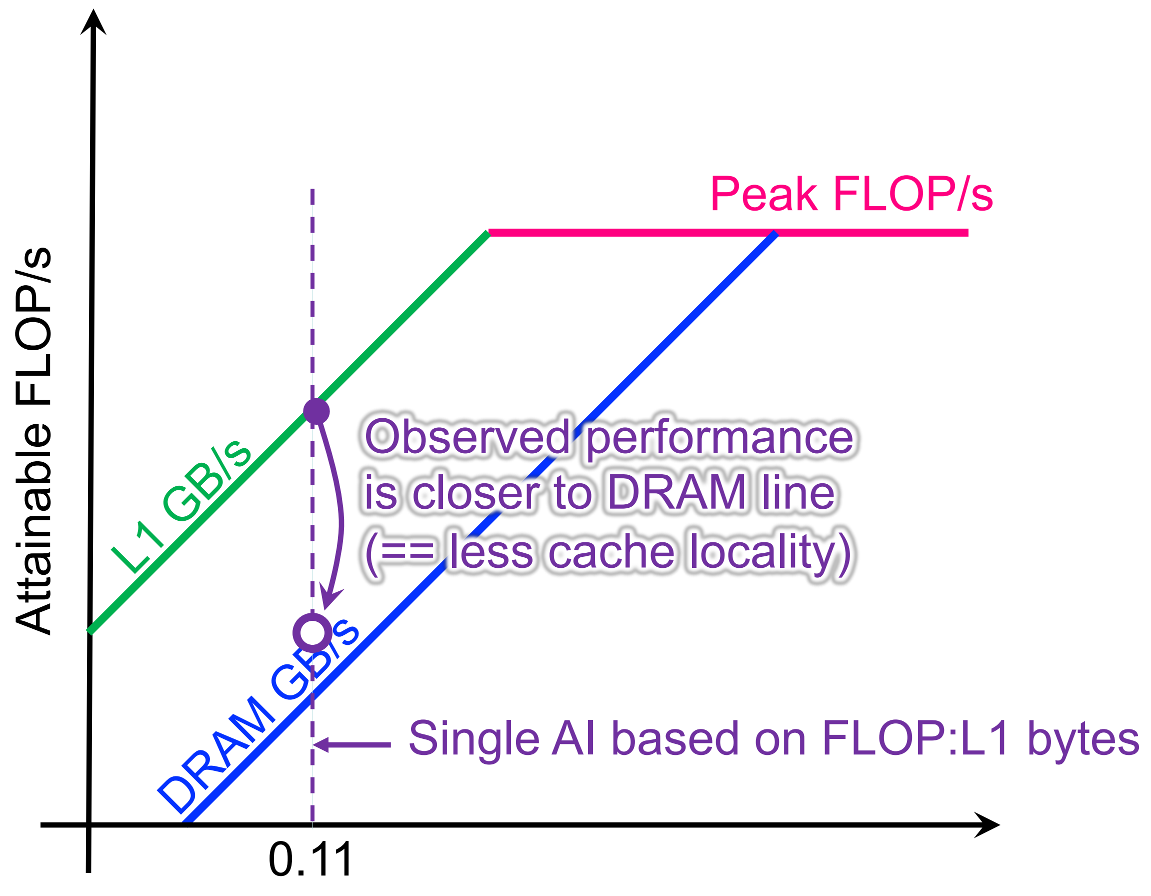
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Arithmetic Intensity (FLOP:Byte)

Cache-Aware Roofline



Arithmetic Intensity (FLOP:Byte)