

Performance Modeling and Analysis

Samuel Williams

Computational Research Division

Lawrence Berkeley National Lab

SWWilliams@lbl.gov

Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This material is based upon work supported by the DOE RAPIDS SciDAC Institute.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- Tuomas Koskela for his creation of the Roofline Advisor demo slides



BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY



Introduction to Performance Modeling

Why Use Performance Models or Tools?

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

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

What are the scaling constants?

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

CGEMM: $O(N^3)$ complexity
where N is the number of rows
(equation)

FFTs: $O(N \log N)$ in the number of

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

MG: $O(N)$ in the number of ele.

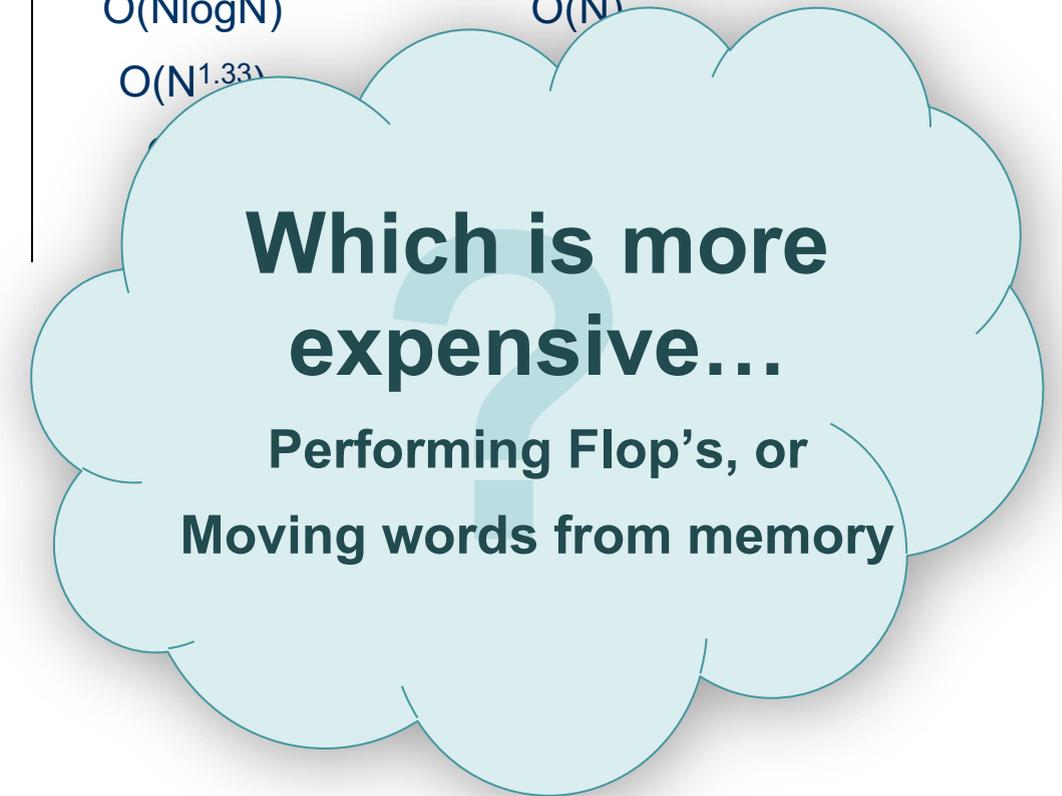
N-body: $O(N^2)$ in the number of

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...
 - Compulsory¹ data movement (array sizes) is a good initial guess...
 - ... but needs refinement for the effects of finite cache capacities

Operation	Flop's	Data
DAXPY	$O(N)$	$O(N)$
DGEMV	$O(N^2)$	$O(N^2)$
DGEMM	$O(N^3)$	$O(N^2)$
FFTs	$O(N \log N)$	$O(N)$
CG	$O(N^{1.33})$	
MG		
N-body		



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

Machine Balance and Arithmetic Intensity

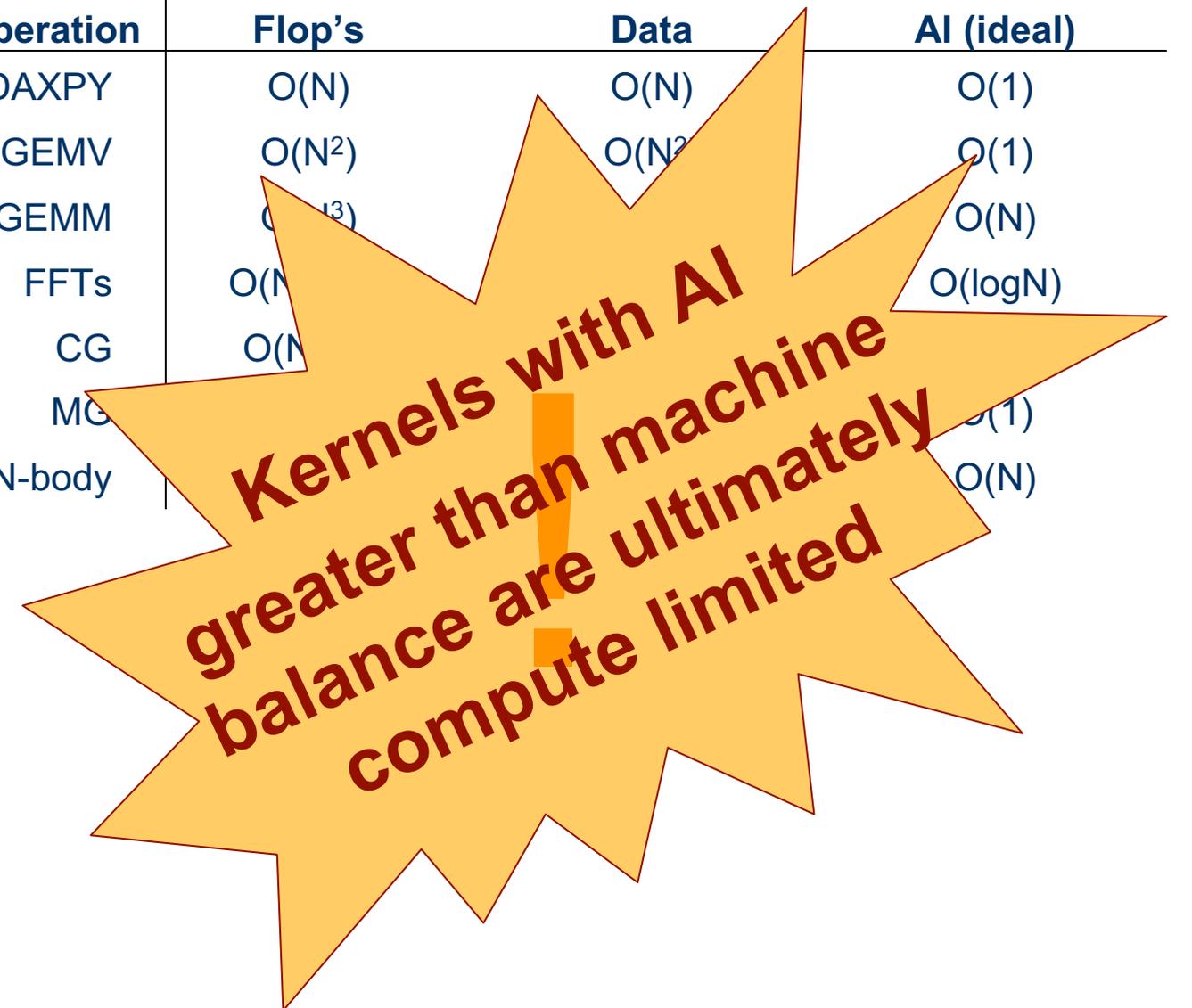
- Data movement and computation can operate at different rates
- We define machine balance as the ratio of...

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

- ...and arithmetic intensity as the ratio of...

$$\text{AI} = \frac{\text{Flop's Performed}}{\text{Data Moved}}$$

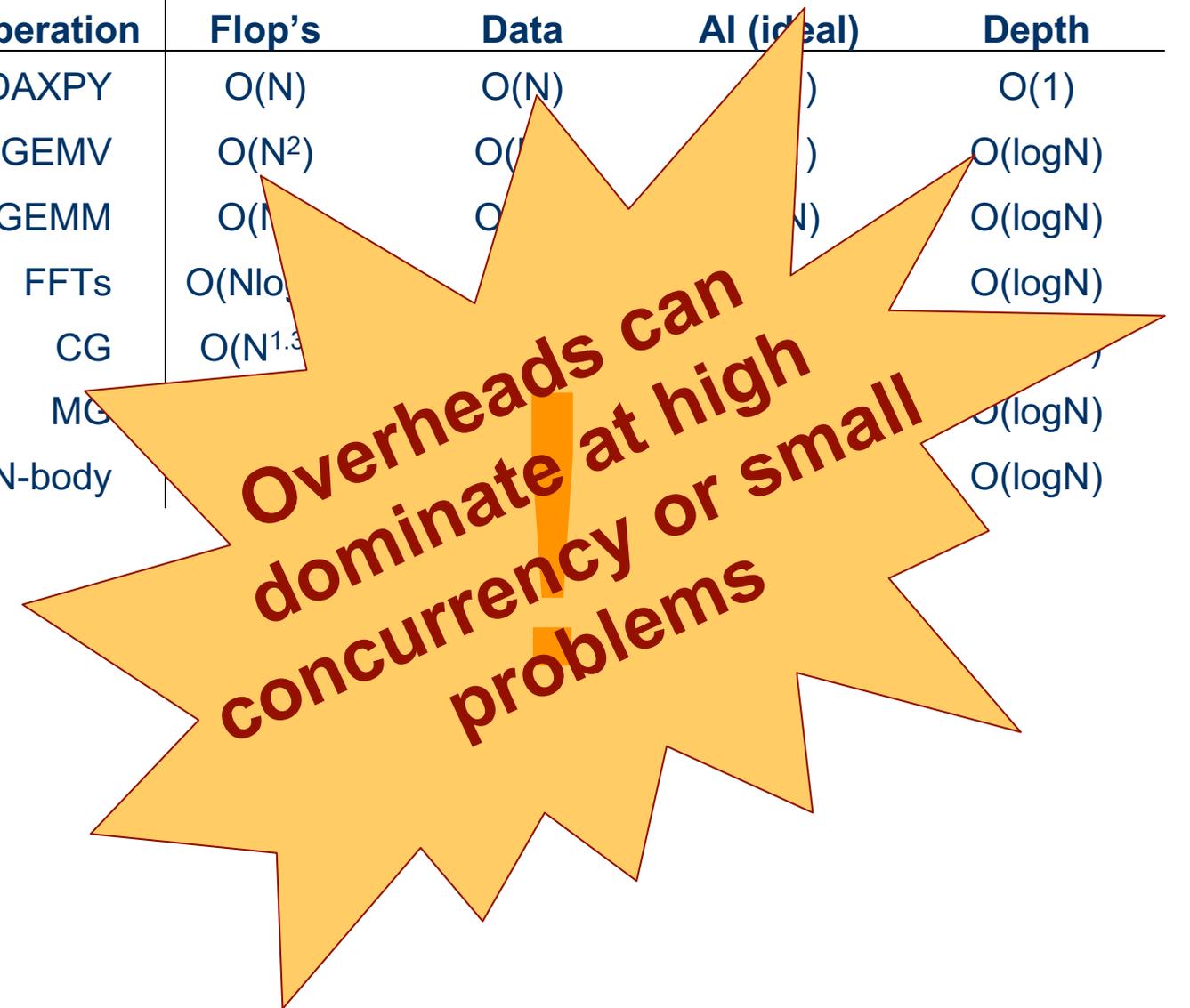
Operation	Flop's	Data	AI (ideal)
DAXPY	$O(N)$	$O(N)$	$O(1)$
DGEMV	$O(N^2)$	$O(N^2)$	$O(1)$
DGEMM	$O(N^3)$	$O(N^2)$	$O(N)$
FFTs	$O(N \log N)$	$O(N)$	$O(\log N)$
CG	$O(N)$	$O(N)$	$O(1)$
MG	$O(N)$	$O(N)$	$O(1)$
N-body	$O(N^2)$	$O(N)$	$O(N)$



Computational Depth

- Sequential CPUs incur latencies and overheads on memory discontinuities and function calls
- Parallel machines incur similar overheads on synchronization (shared memory), point-to-point communication, reductions, and broadcasts.
- Thus, we can classify algorithms by **depth** (max depth of the algorithm's dependency chain)

Operation	Flop's	Data	AI (ideal)	Depth
DAXPY	$O(N)$	$O(N)$	$O(N)$	$O(1)$
DGEMV	$O(N^2)$	$O(N)$	$O(N)$	$O(\log N)$
DGEMM	$O(N^3)$	$O(N^2)$	$O(N^2)$	$O(\log N)$
FFTs	$O(N \log N)$	$O(N)$	$O(N)$	$O(\log N)$
CG	$O(N^{1.5})$	$O(N)$	$O(N)$	$O(\log N)$
MG	$O(N)$	$O(N)$	$O(N)$	$O(\log N)$
N-body	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(\log N)$



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...

**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	Roofline Model
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
LogP	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

LogGP

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
LogCA	PCIe data movement	PCIe bandwidth
	Depth	OMP C
	MPI Message Size	Network B
	MPI Send:Wait ratio	Network Gap
	#MPI Wait's	Network Latency

Think about which model to use



BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY



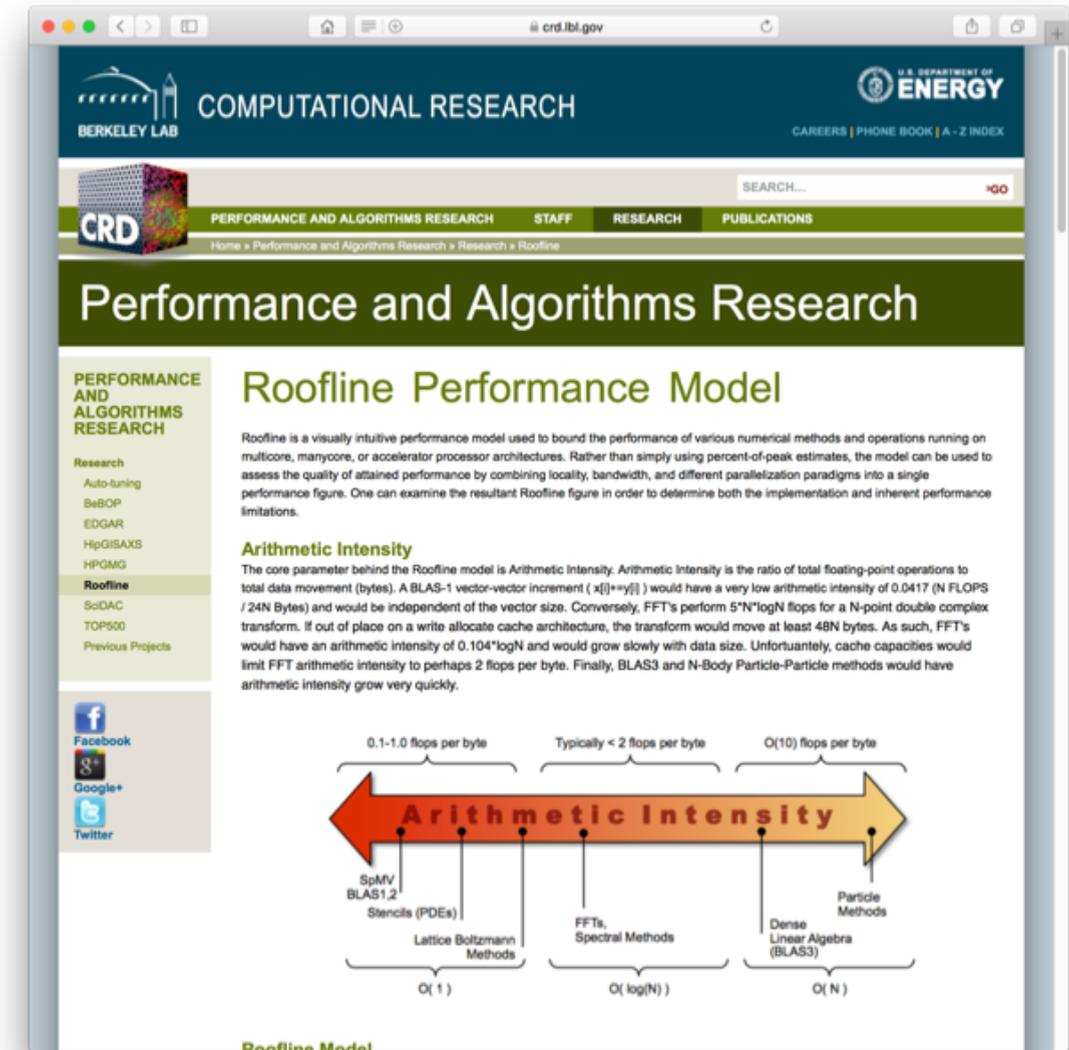
Introduction to the Roofline Model

Performance Models / Simulators

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

Roofline Model

- **Roofline Model** is a throughput-oriented performance model...
 - Tracks rates not times
 - Augmented with Little's Law
(concurrency = latency*bandwidth)
 - Independent of ISA and architecture (applies to CPUs, GPUs, Google TPUs¹, etc...)



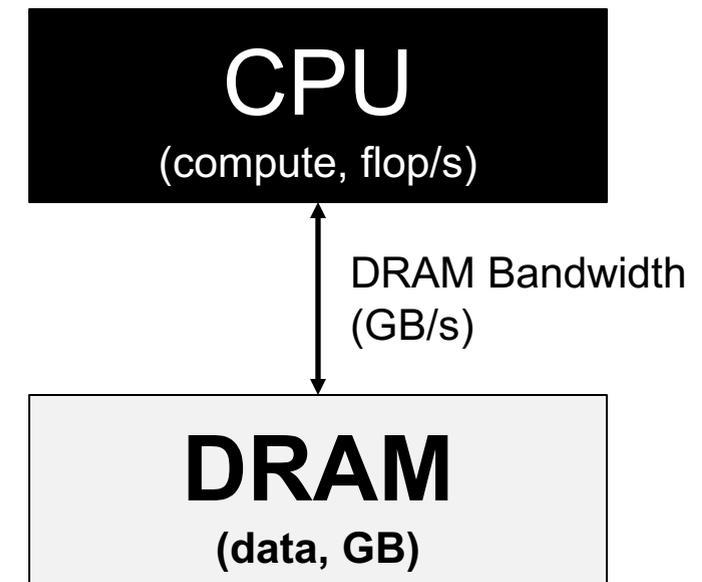
<https://crd.lbl.gov/departments/computer-science/PAR/research/roofline>

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

(DRAM) Roofline

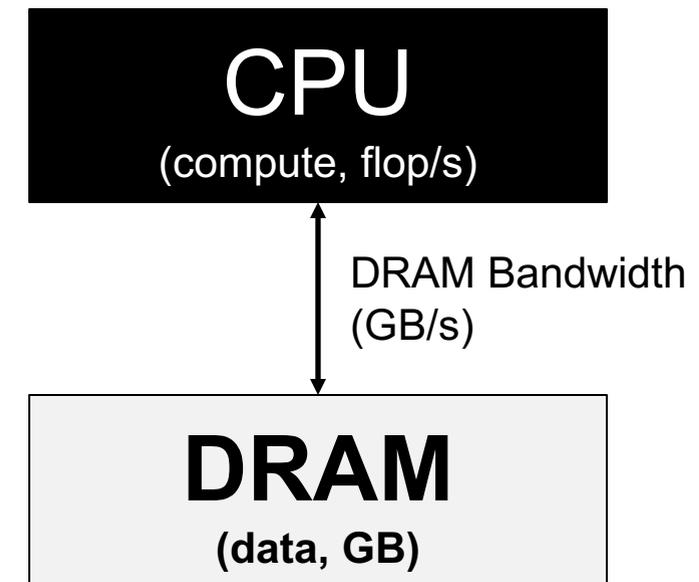
- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
 - Idealized processor/caches
 - Cold start (data in DRAM)

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



(DRAM) Roofline

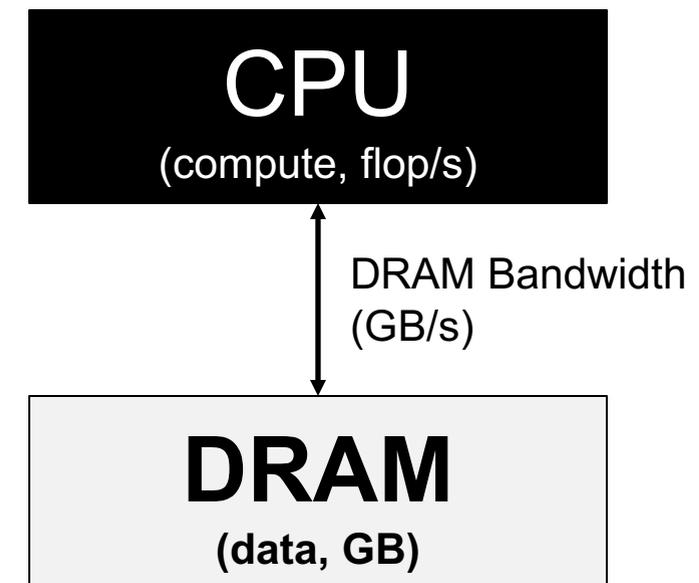
- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
 - Idealized processor/caches
 - Cold start (data in DRAM)



$$\frac{\text{Time}}{\text{\#FP ops}} = \max \left\{ \begin{array}{l} 1 / \text{Peak GFlop/s} \\ \text{\#Bytes} / \text{\#FP ops} / \text{Peak GB/s} \end{array} \right.$$

(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)



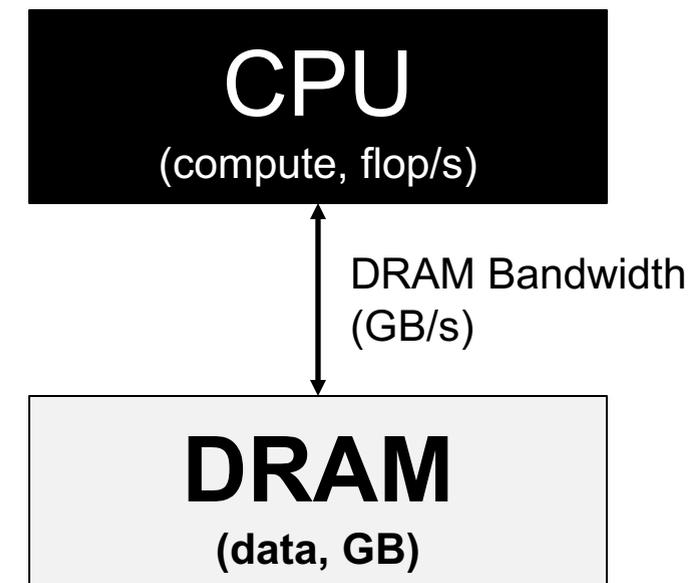
$$\frac{\#FP\ ops}{Time} = \min \left\{ \begin{array}{l} \text{Peak GFlop/s} \\ (\#FP\ ops / \#Bytes) * \text{Peak GB/s} \end{array} \right.$$

(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)

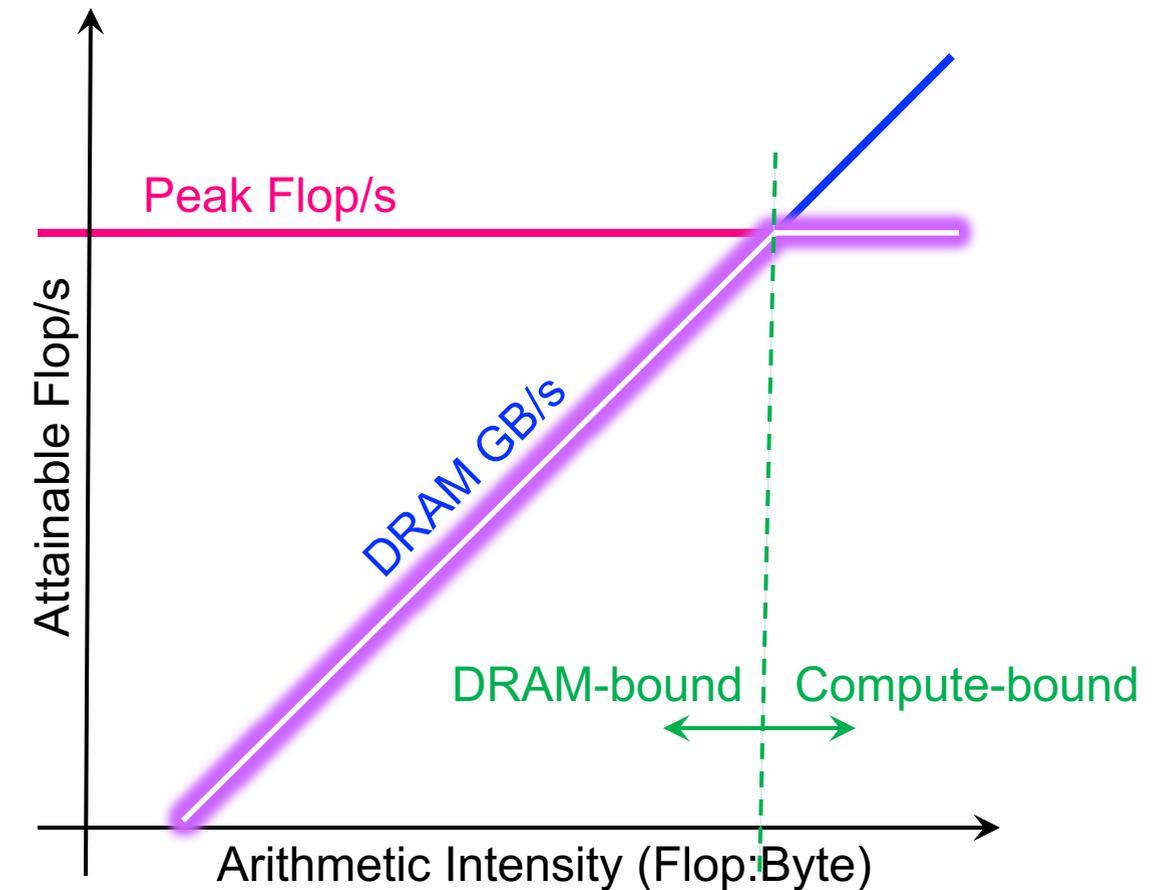
$$\text{GFlop/s} = \min \left\{ \begin{array}{l} \text{Peak GFlop/s} \\ \text{AI} * \text{Peak GB/s} \end{array} \right.$$

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



(DRAM) Roofline

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



Roofline Example #1

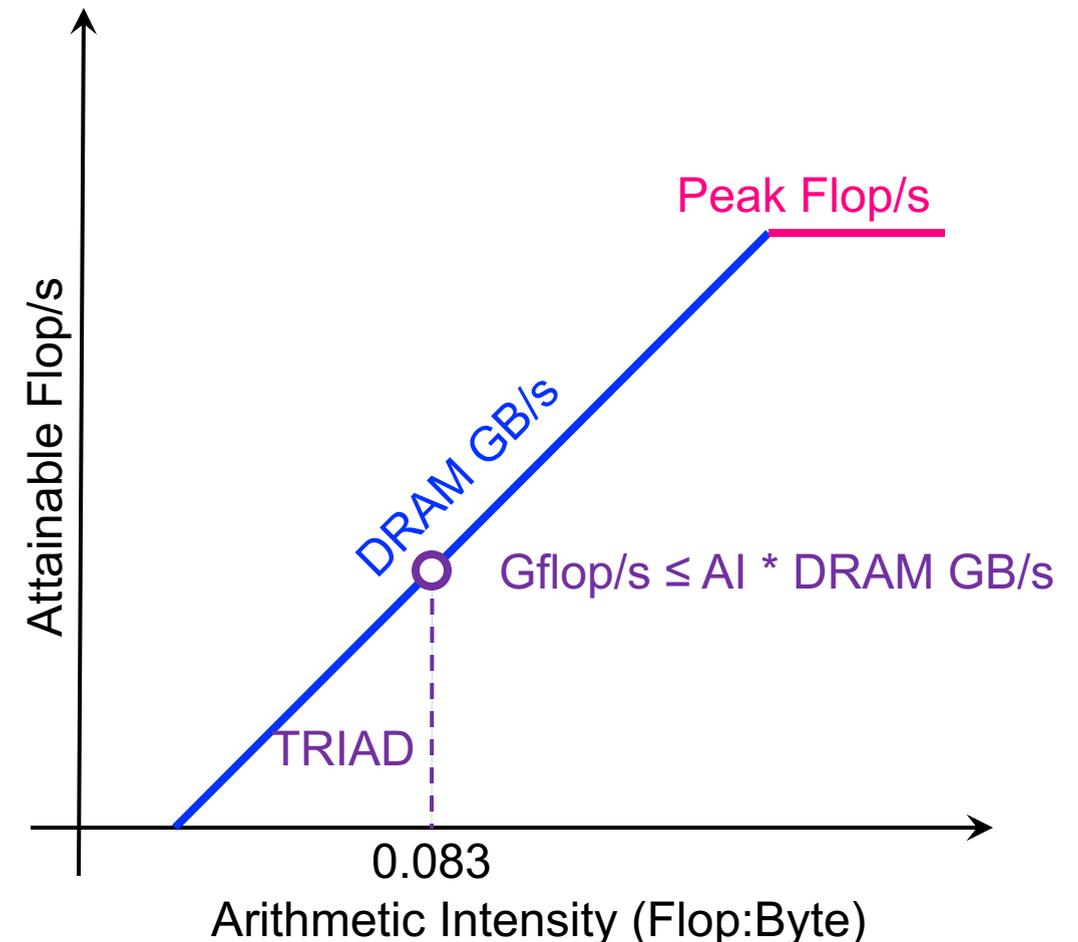
- Typical machine balance is 5-10 flops per byte...

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

- Consider STREAM Triad...

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

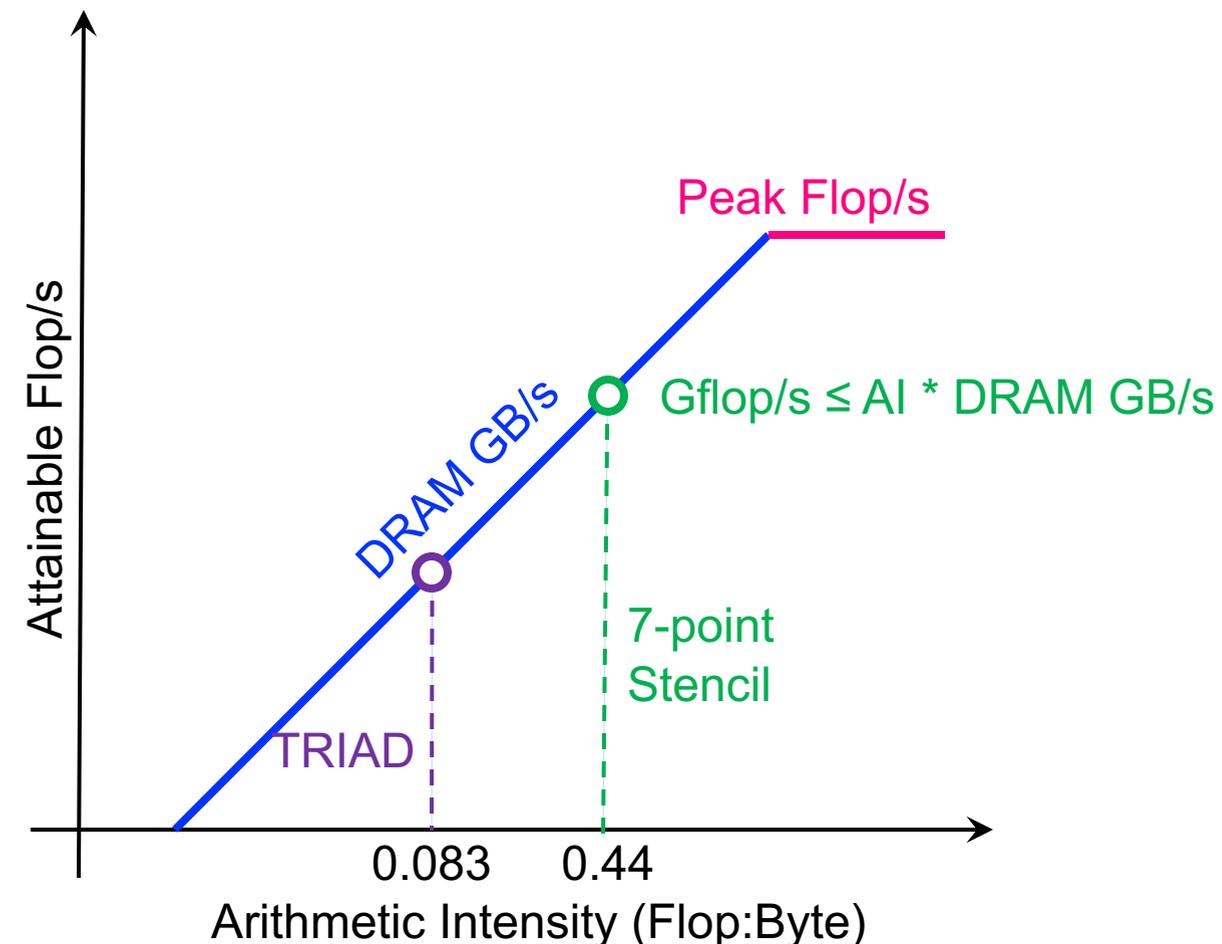
- 2 flops per iteration
- Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
- **AI = 0.083 flops per byte == Memory bound**



Roofline Example #2

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

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

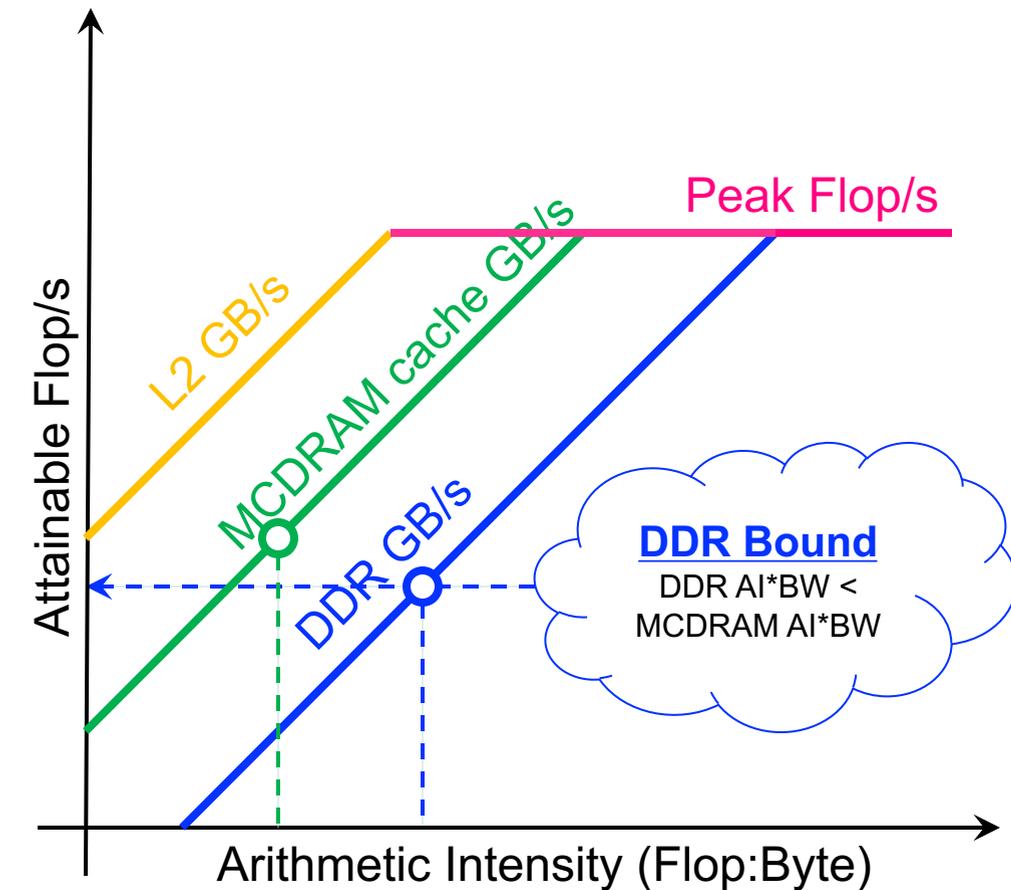


Hierarchical Roofline

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

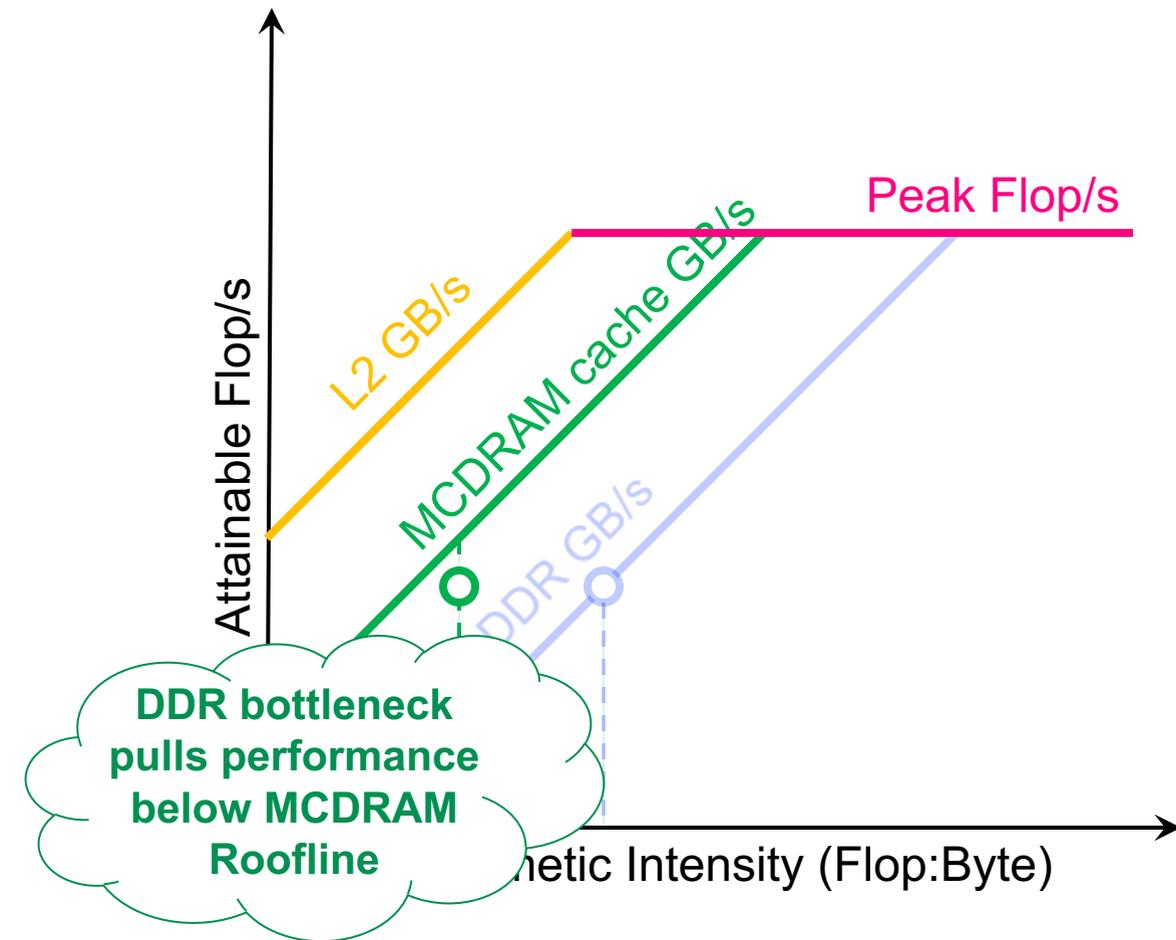
Hierarchical Roofline

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



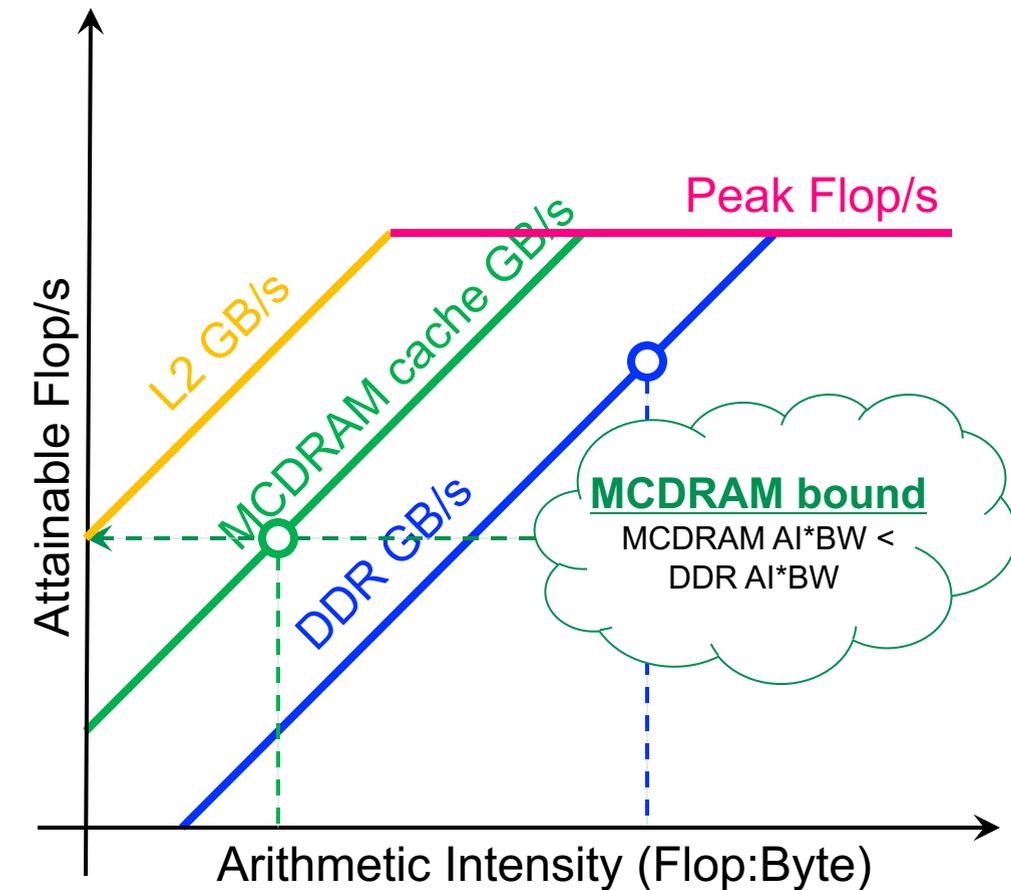
Hierarchical Roofline

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



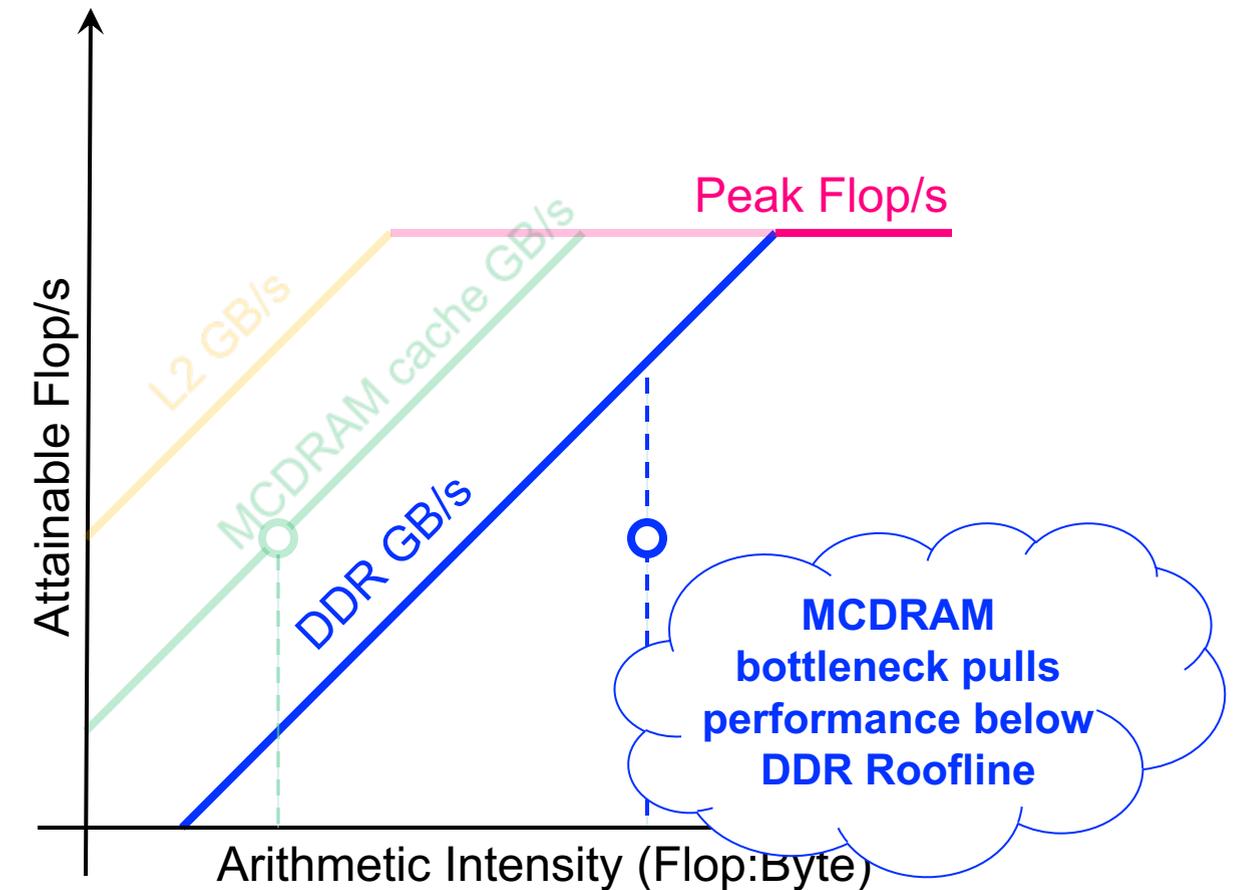
Hierarchical Roofline

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



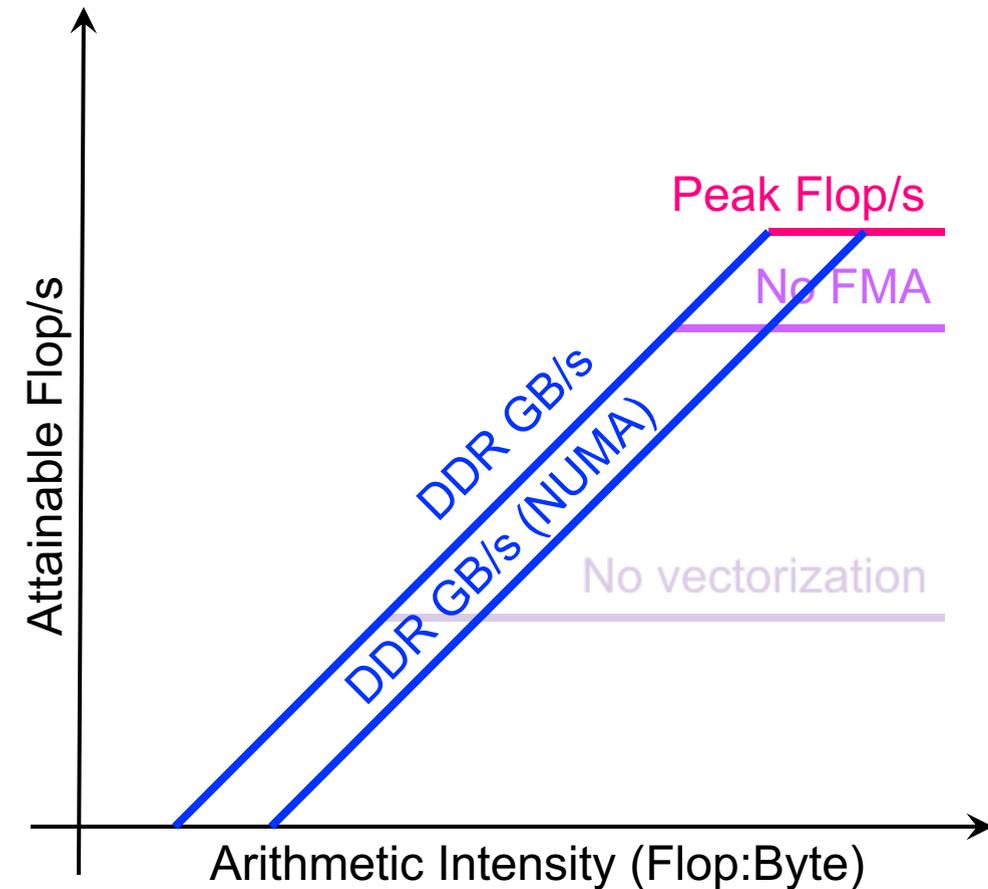
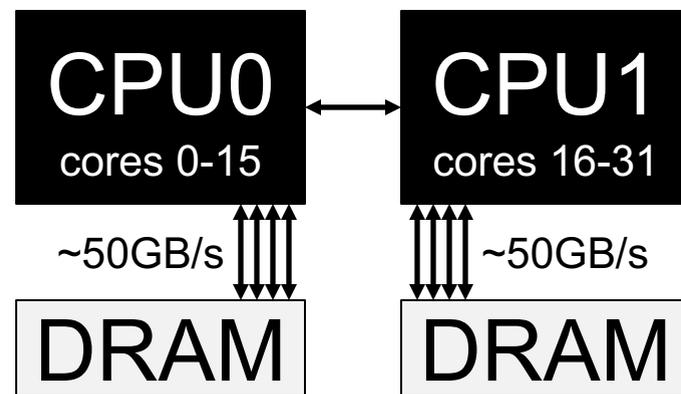
Hierarchical Roofline

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



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





Modeling In-Core Performance Effects

Data, Instruction, Thread-Level Parallelism...

- Modern CPUs use several techniques to increase per core Flop/s

Fused Multiply Add

- $w = x*y + z$ is a common idiom in linear algebra
- Rather than using separate multiply and add instructions, use a single instruction (FMA)
- The hardware chains the multiply and add in a single pipeline so that it can complete FMA/cycle

Resurgence...
Tensor Cores,
QFMA, etc...

Vector Instructions

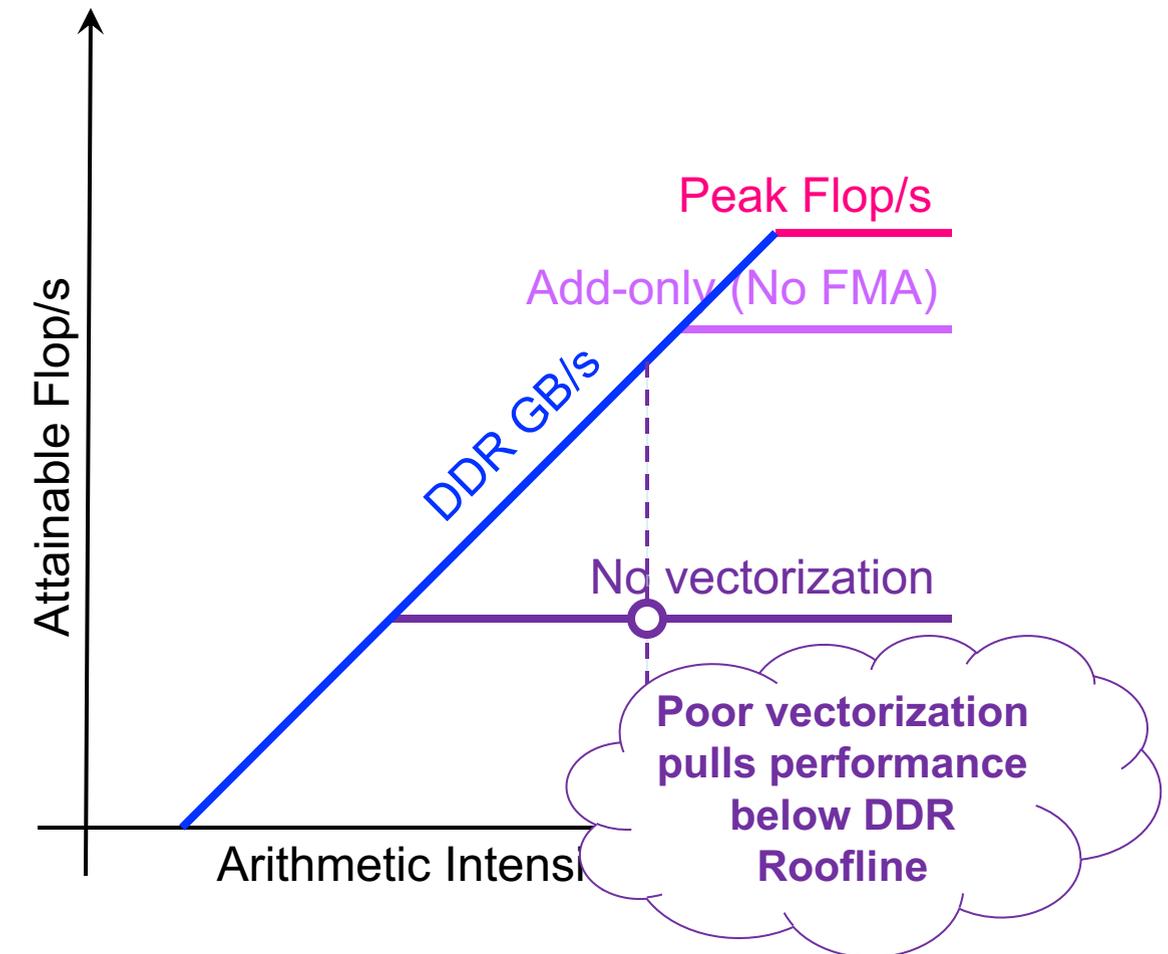
- Many HPC codes apply the same operation to a vector of elements
- Vendors provide vector instructions that apply the same operation to 2, 4, 8, 16 elements...
 $x [0:7] *y [0:7] + z [0:7]$
- Vector FPUs complete 8 vector operations/cycle

Deep pipelines

- The hardware for a FMA is substantial.
- Breaking a single FMA up into several smaller operations and pipelining them allows vendors to increase GHz
- Little's Law applies...
need $FP_Latency * FP_bandwidth$ independent instructions

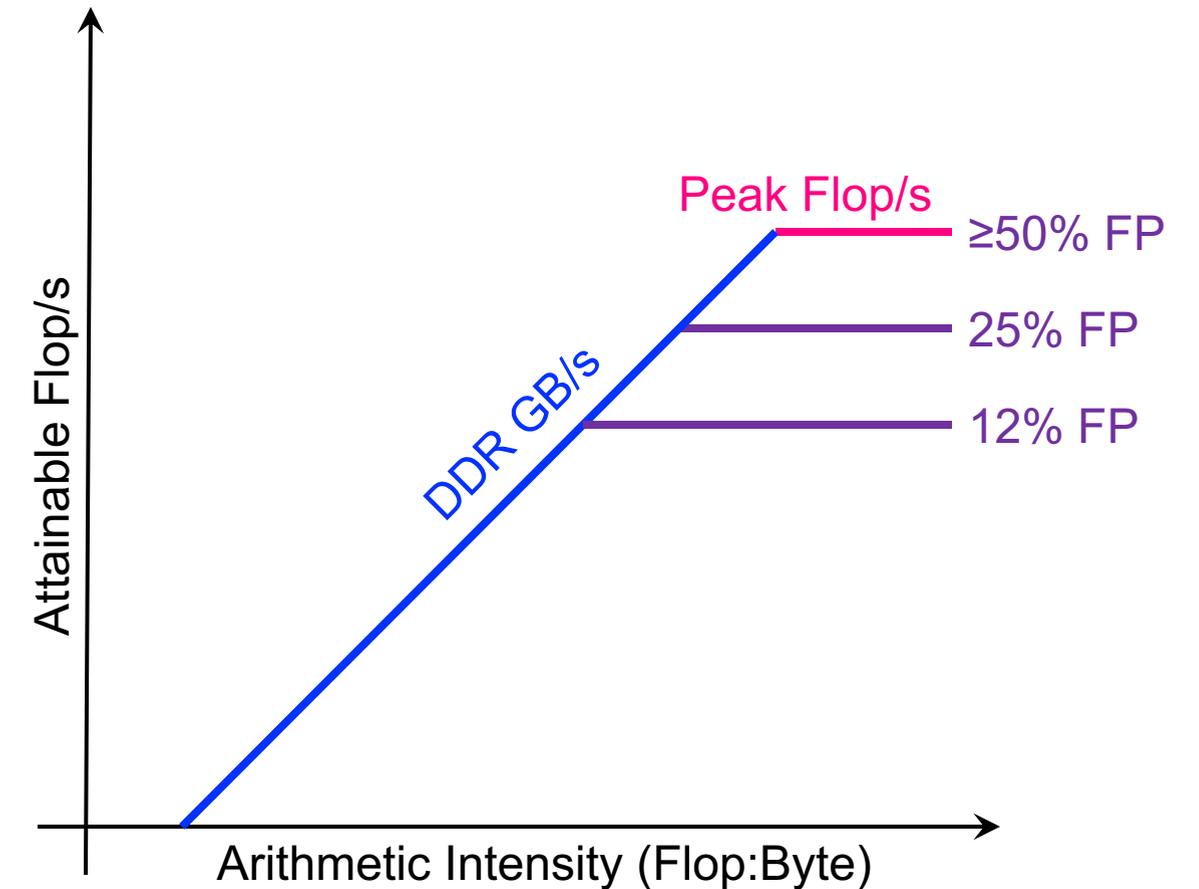
Data, Instruction, Thread-Level Parallelism...

- If every instruction were an ADD (instead of FMA), **performance would drop by 2x on KNL or 4x on Haswell**
- Similarly, if one had no vector instructions, performance would drop by **another 8x on KNL and 4x on Haswell**
- FP Divides can be even worse.
- Lack of threading will reduce performance by 64x on KNL.



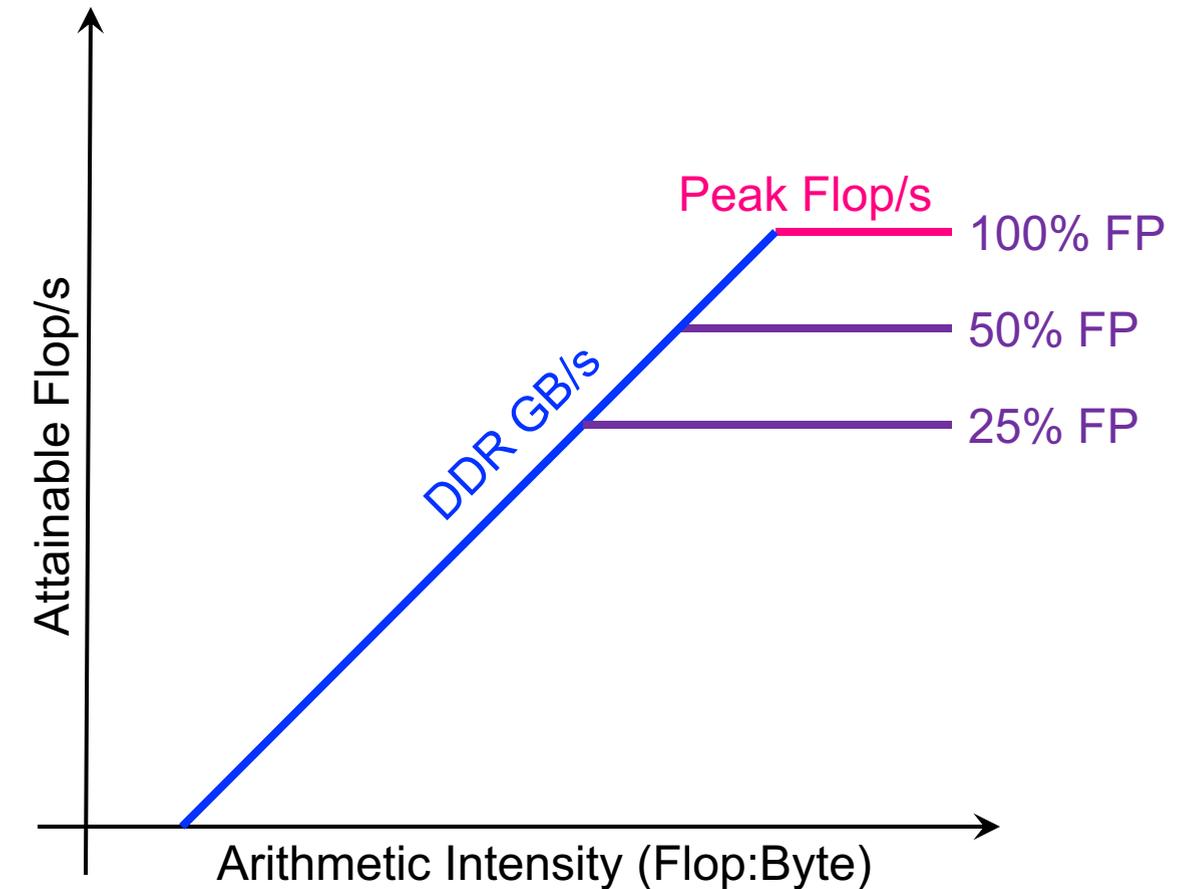
Superscalar vs. instruction mix

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



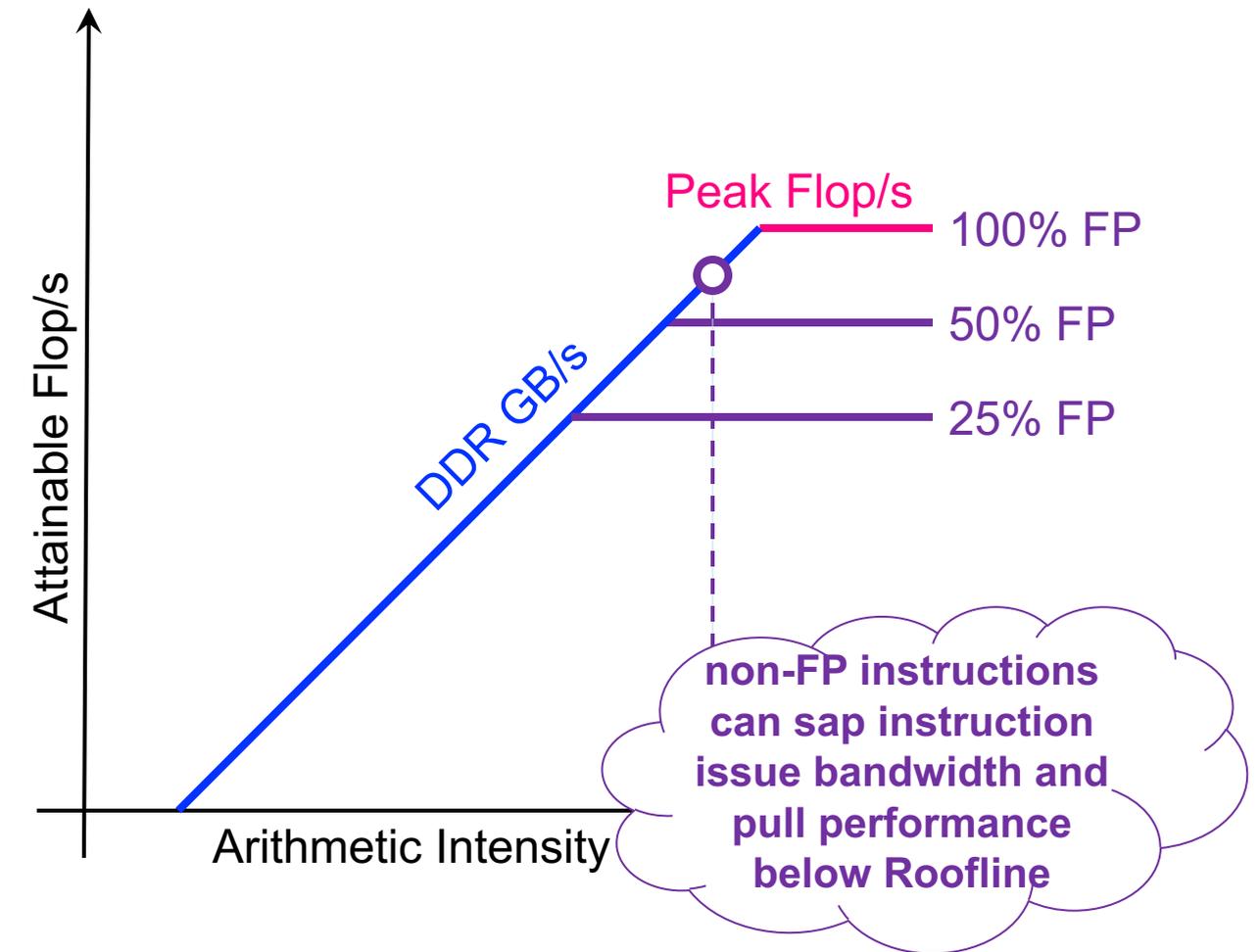
Superscalar vs. instruction mix

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



Superscalar vs. instruction mix

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





BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

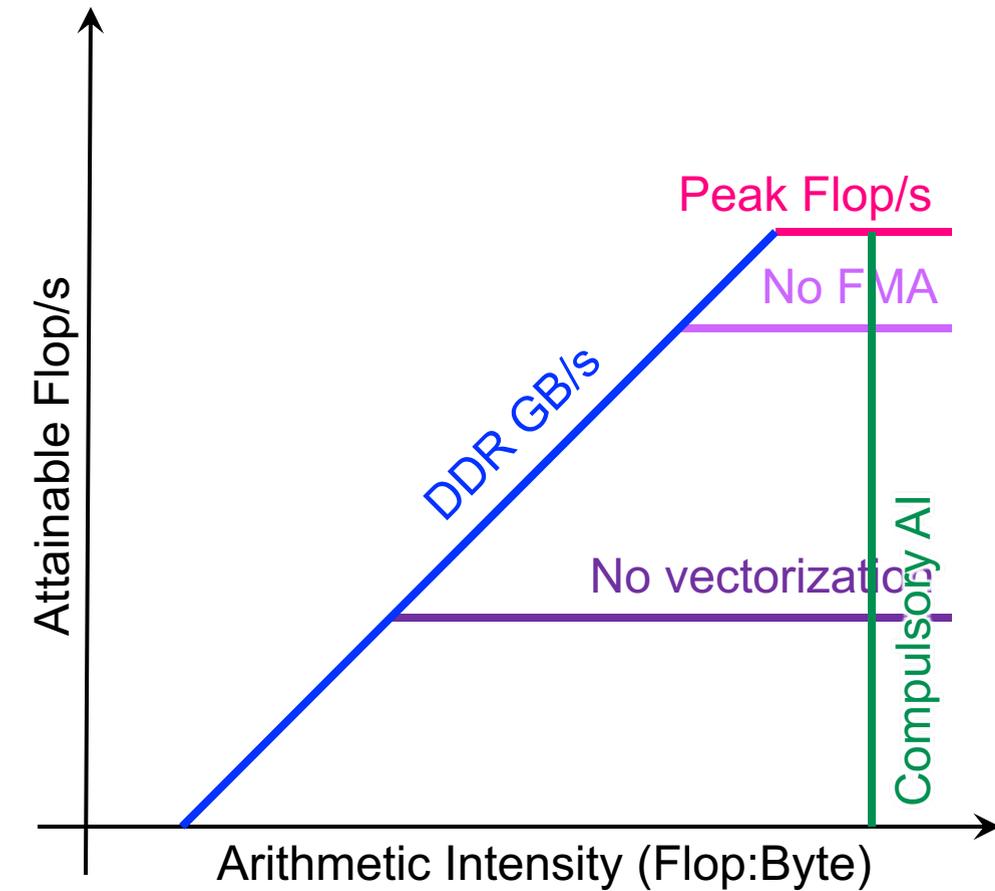


U.S. DEPARTMENT OF
ENERGY

Modeling Cache Effects

Locality Walls

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

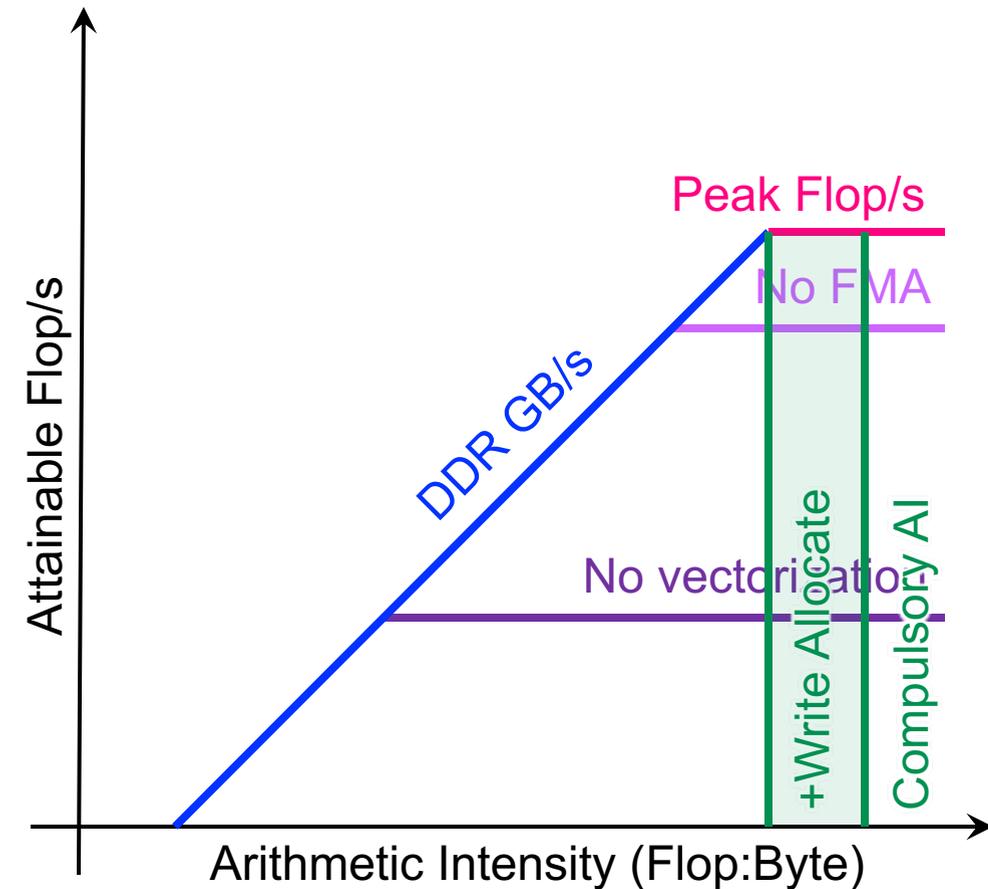


$$AI = \frac{\#Flop's}{\text{Compulsory Misses}}$$

Locality Walls

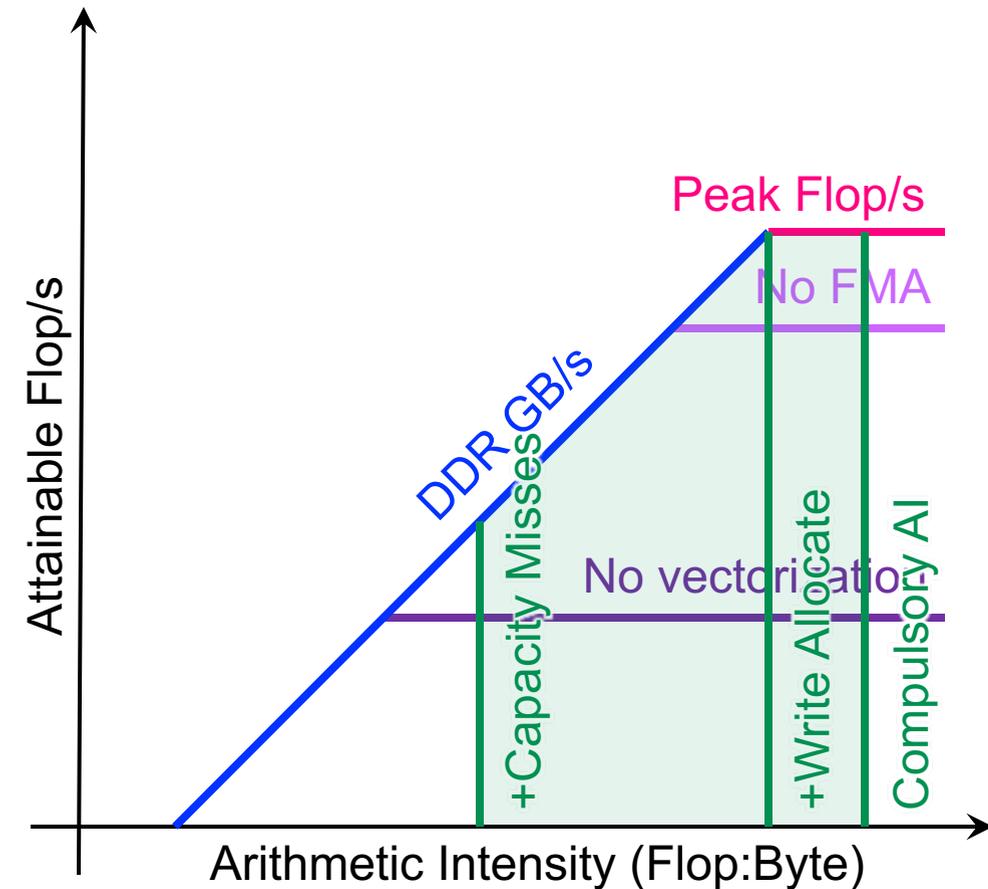
- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI

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



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

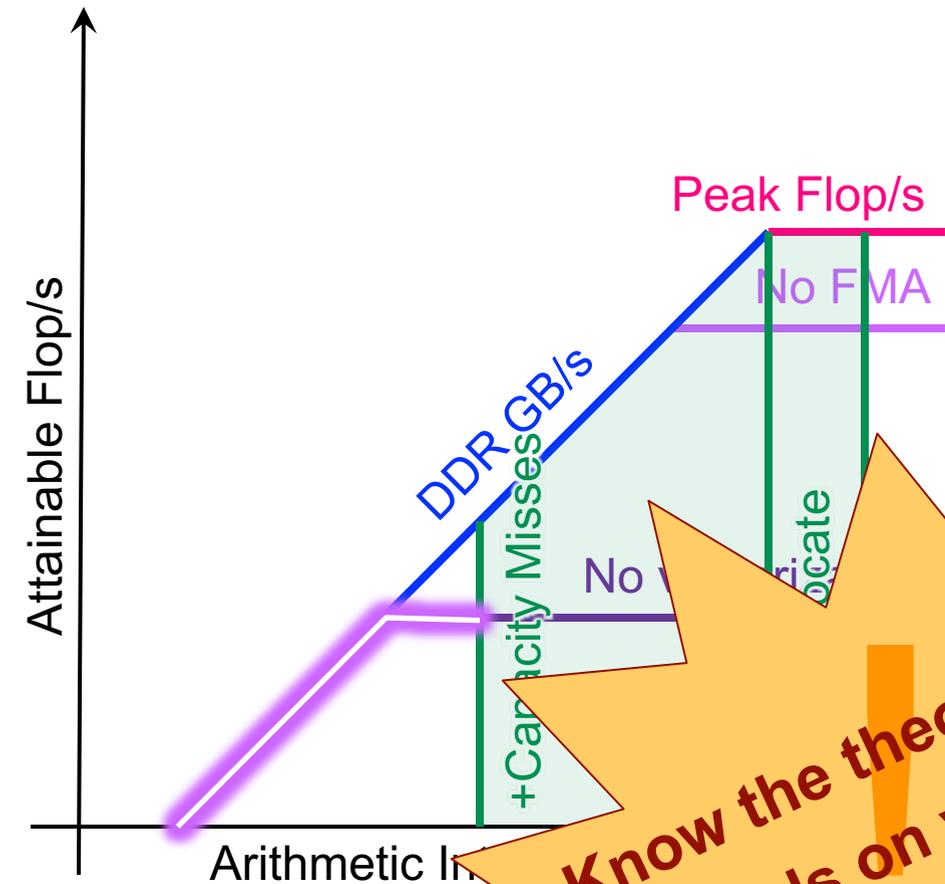


$$AI = \frac{\#Flop's}{\text{Compulsory Misses} + \text{Write Allocates} + \text{Capacity Misses}}$$

Locality Walls

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

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



Know the theoretical bounds on your AI.



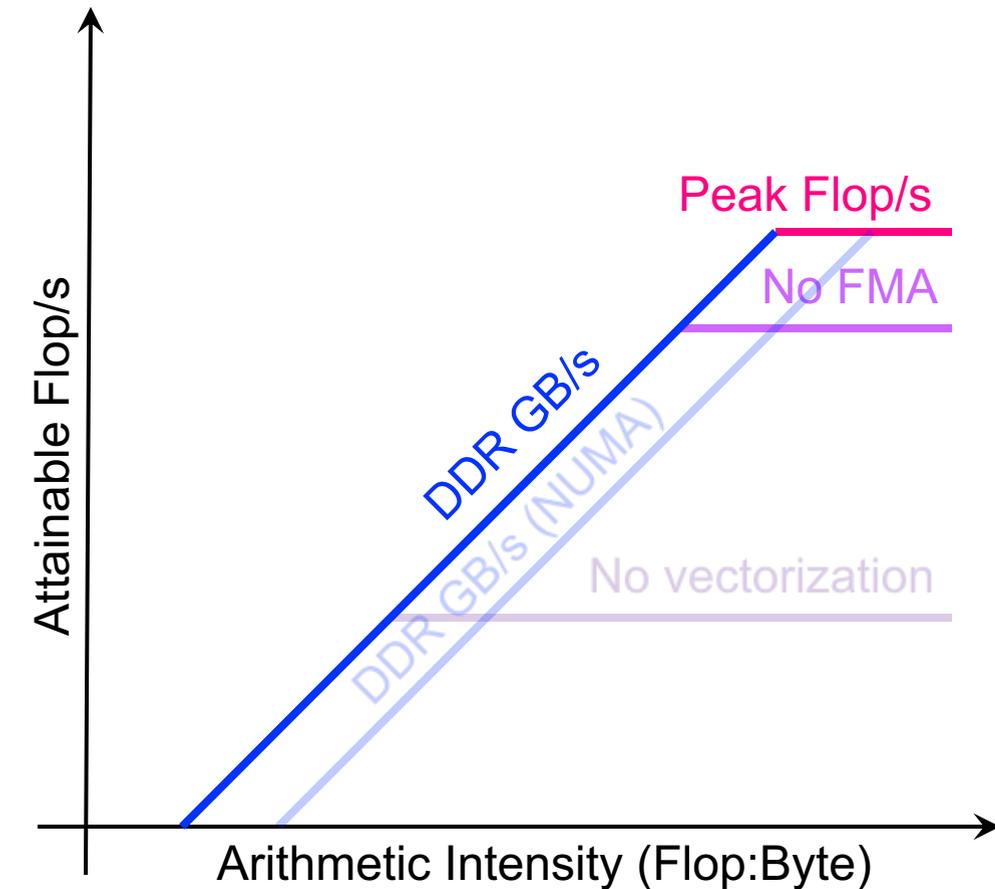
BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY



General Strategy Guide

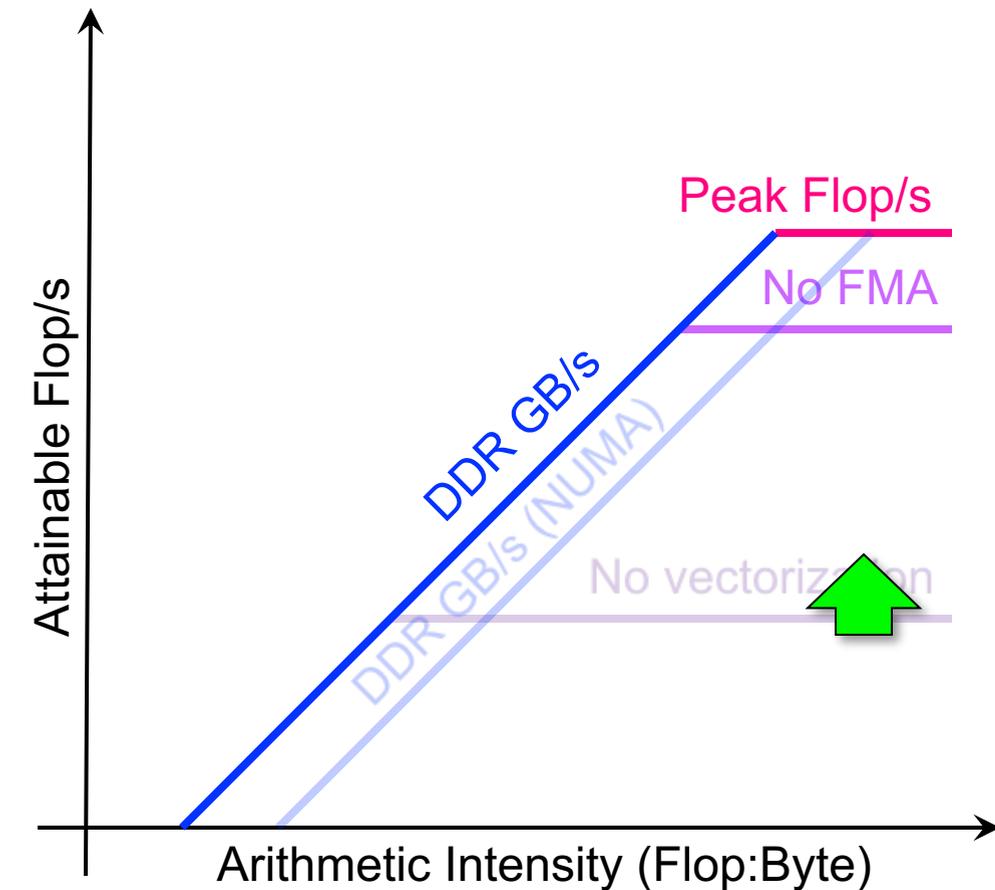
General Strategy Guide

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



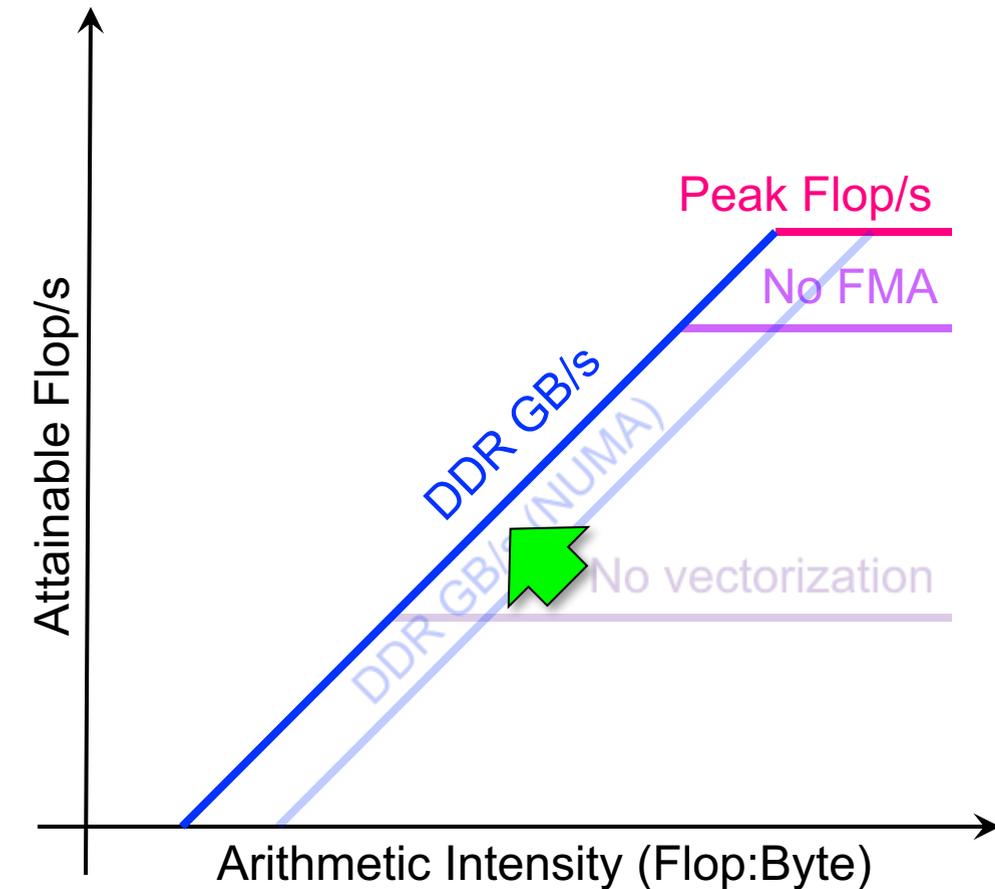
General Strategy Guide

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



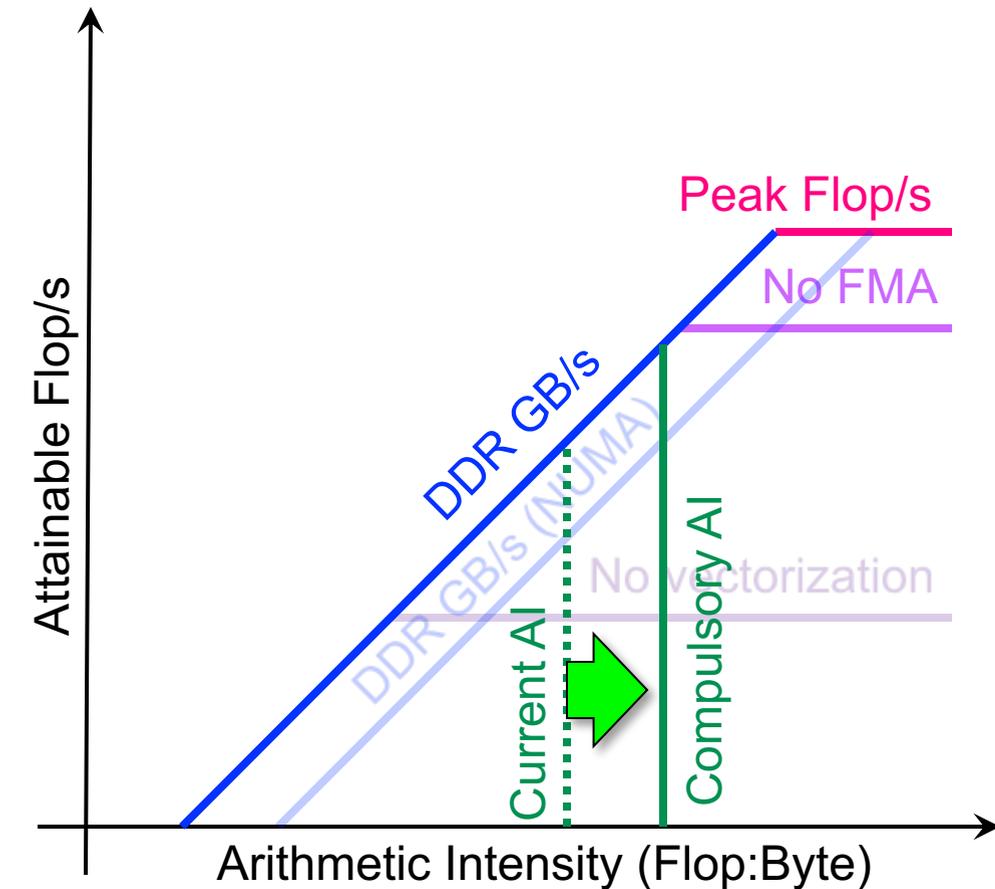
General Strategy Guide

- 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 allocation)**



General Strategy Guide

- 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 allocation)
- **Minimize data movement (increase AI)**





Constructing a Roofline Model requires answering some questions...

Questions can overwhelm users...

Properties of the target machine

(Benchmarking)

What is my machine's peak flop/s?

How important is FMA on my machine?

What is my machine's DDR GB/s?
L2 GB/s?

Properties of an application's execution

(Instrumentation)

How much data did my kernel actually move?

How many flop's did my kernel actually do?

How much did that divide hurt?

Fundamental properties of the kernel constrained by hardware

(Theory)

What is my kernel's compulsion AI? (communication lower bounds)

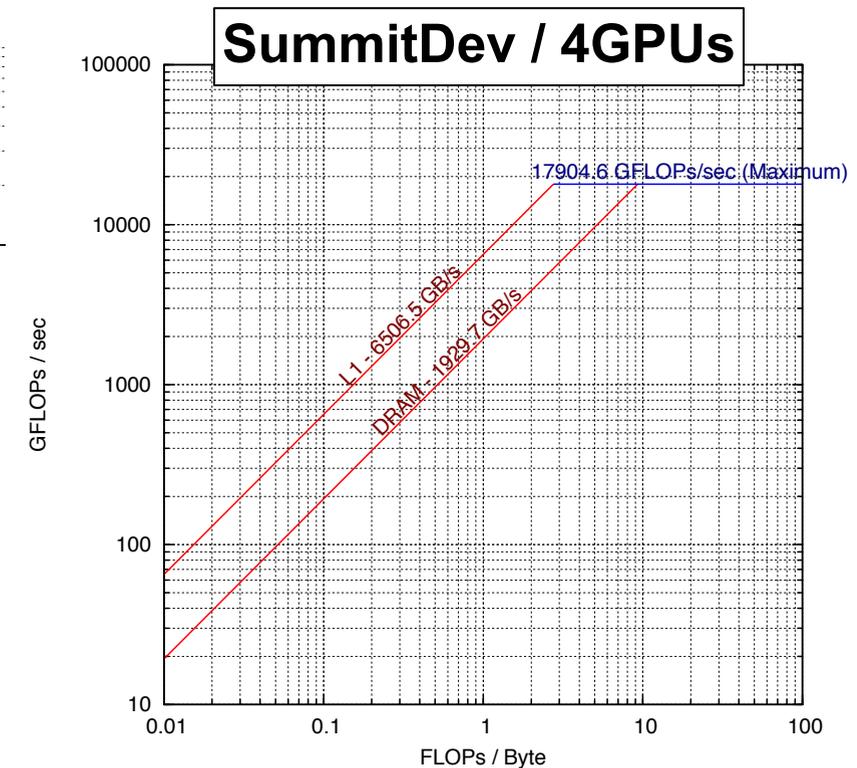
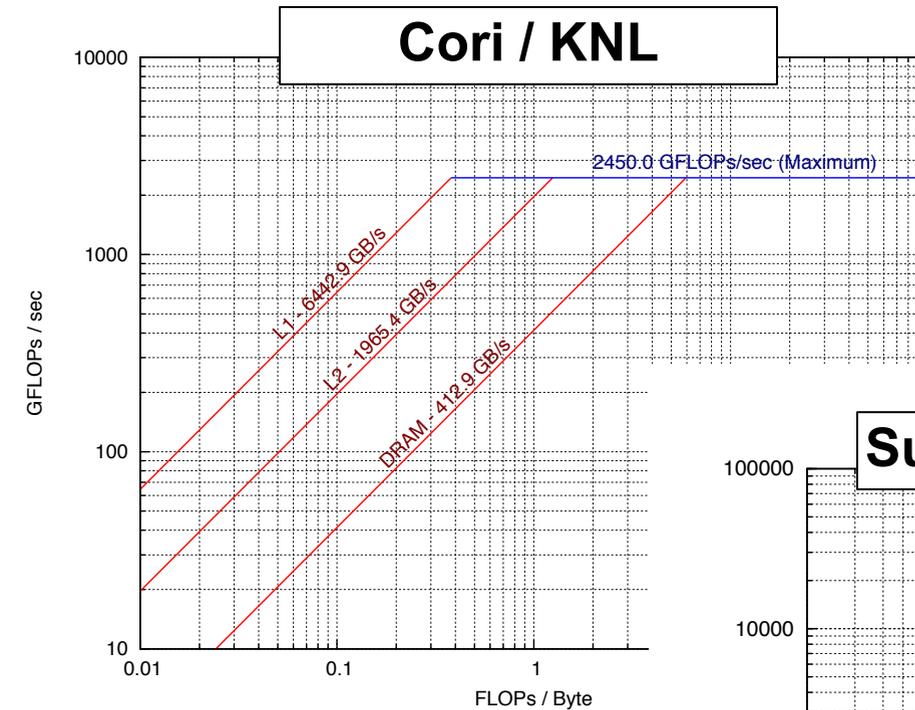
Can my kernel ever be vectorized?



**To answer these
questions, we need
tools...**

Node Characterization?

- “Marketing Numbers” can be deceptive...
 - Pin BW vs. real 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**

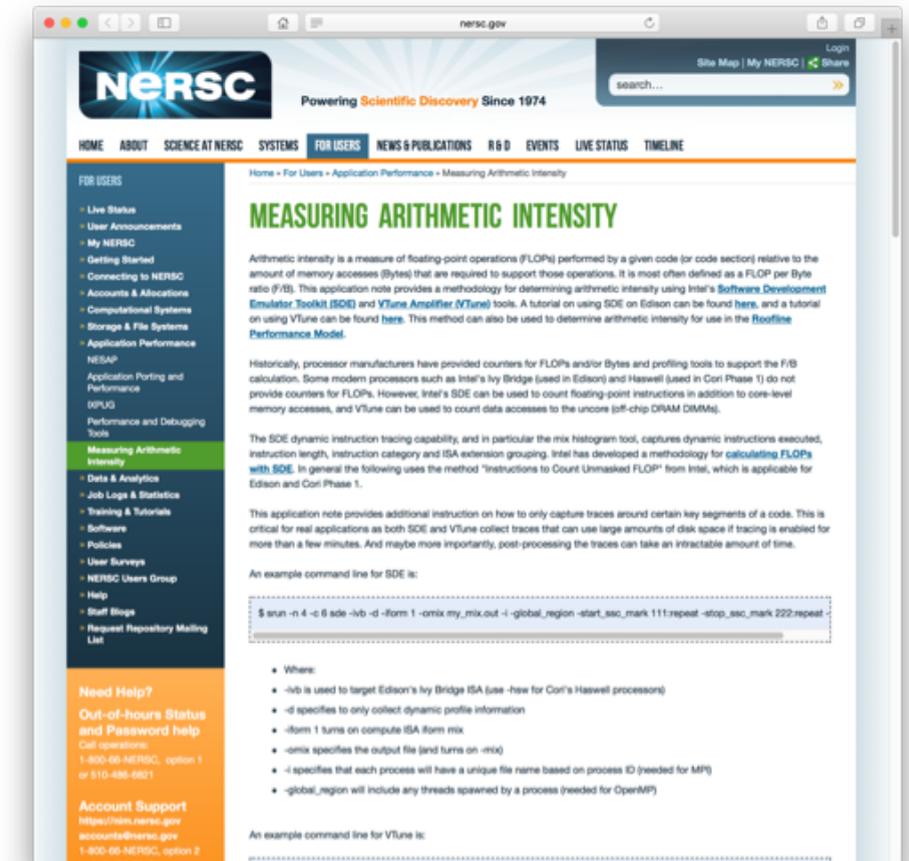


Instrumentation with Performance Counters?

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

Forced to Cobble Together Tools...

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



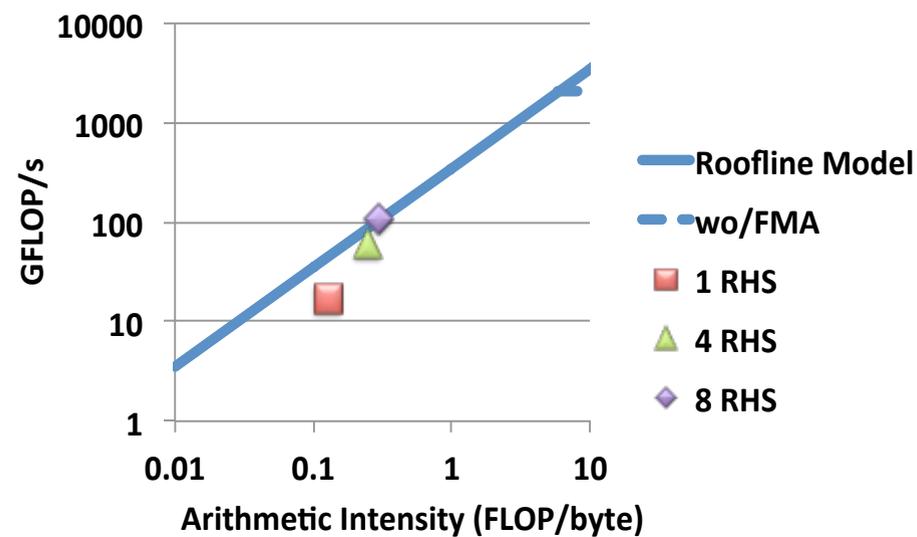
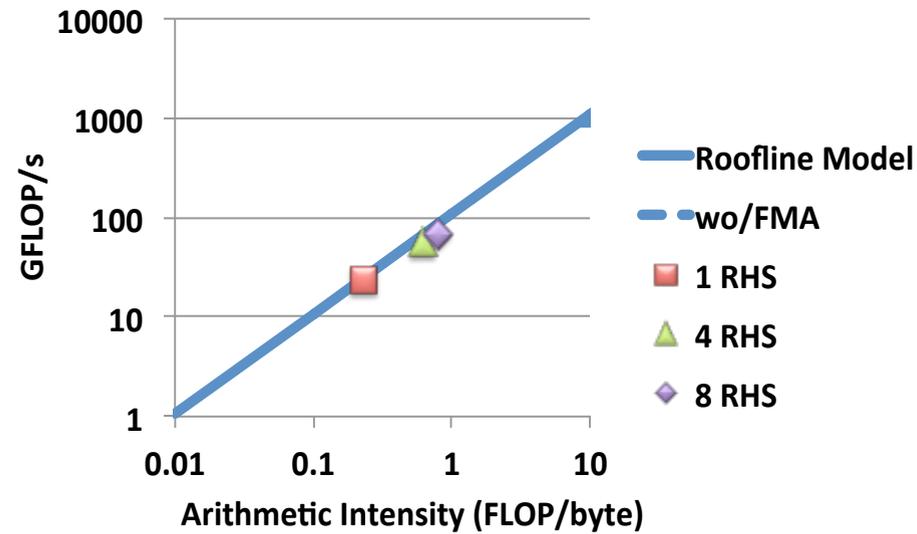
<http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/>

Initial Roofline Analysis of NESAP Codes

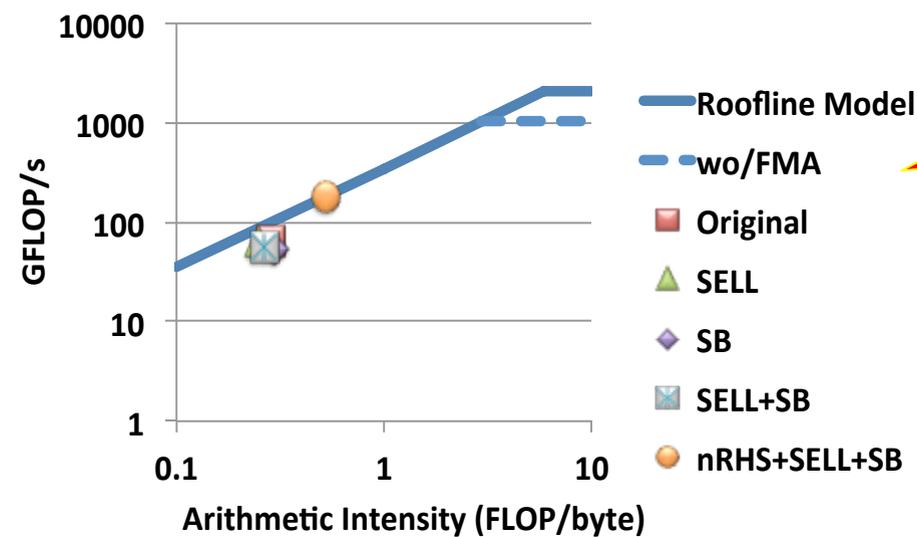
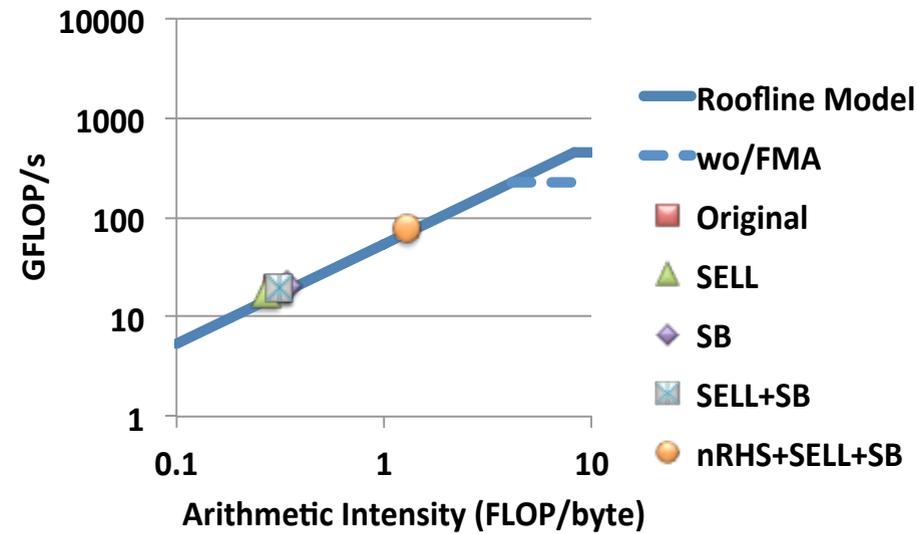
2P HSW

KNL

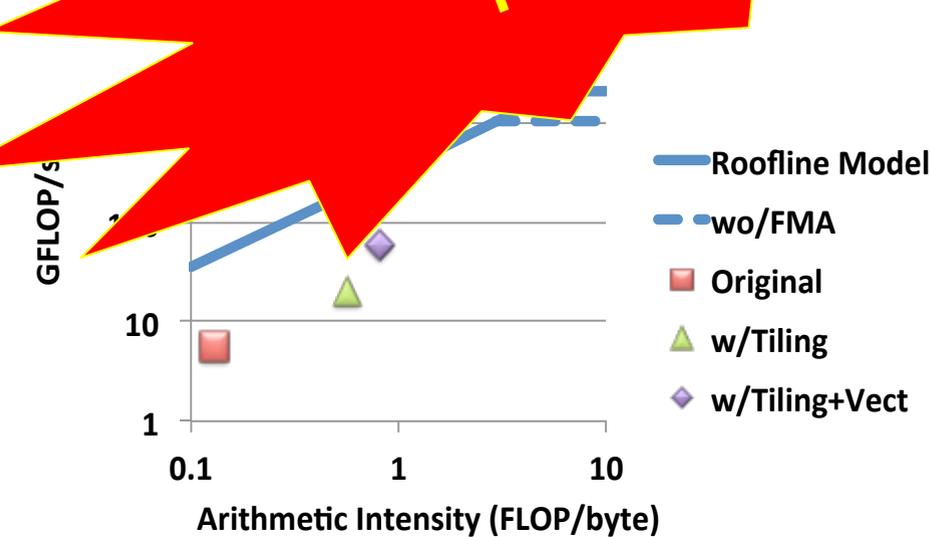
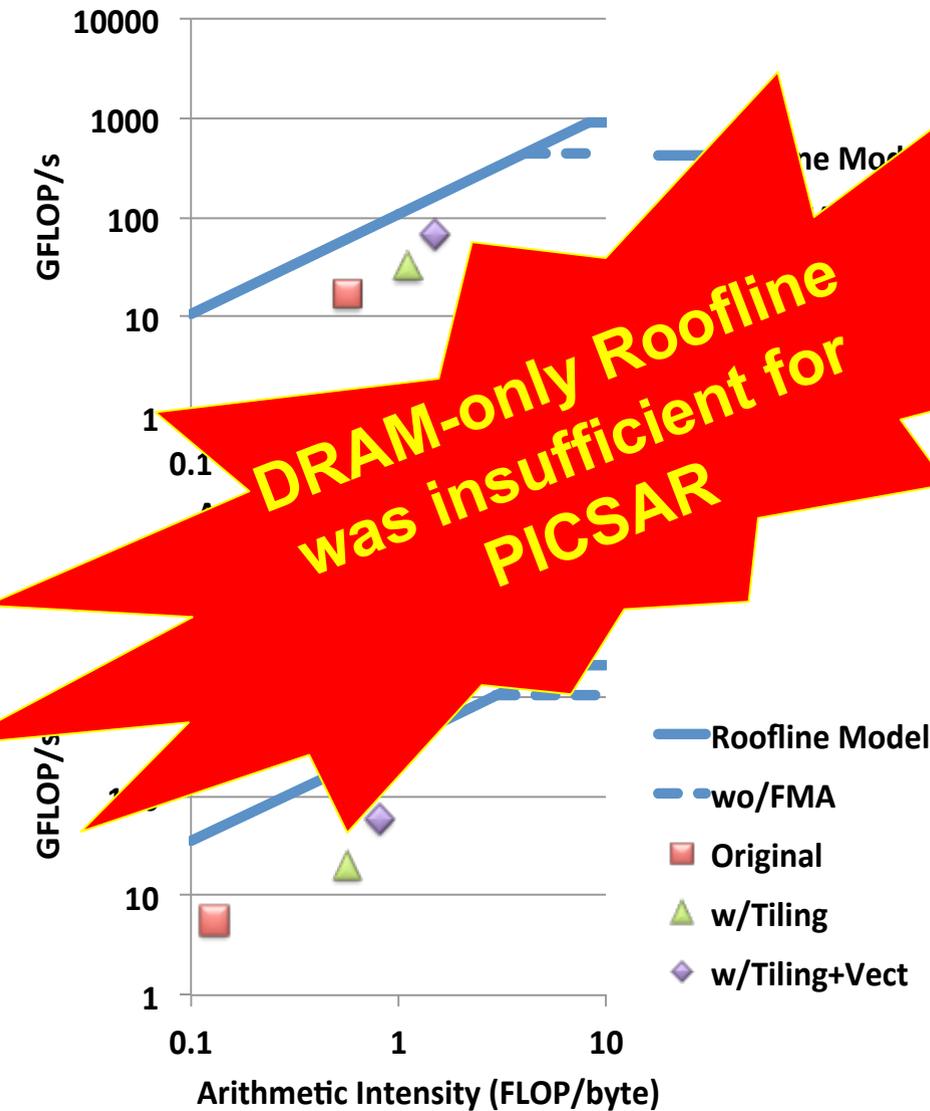
MFDn



EMGeo



PICSAR

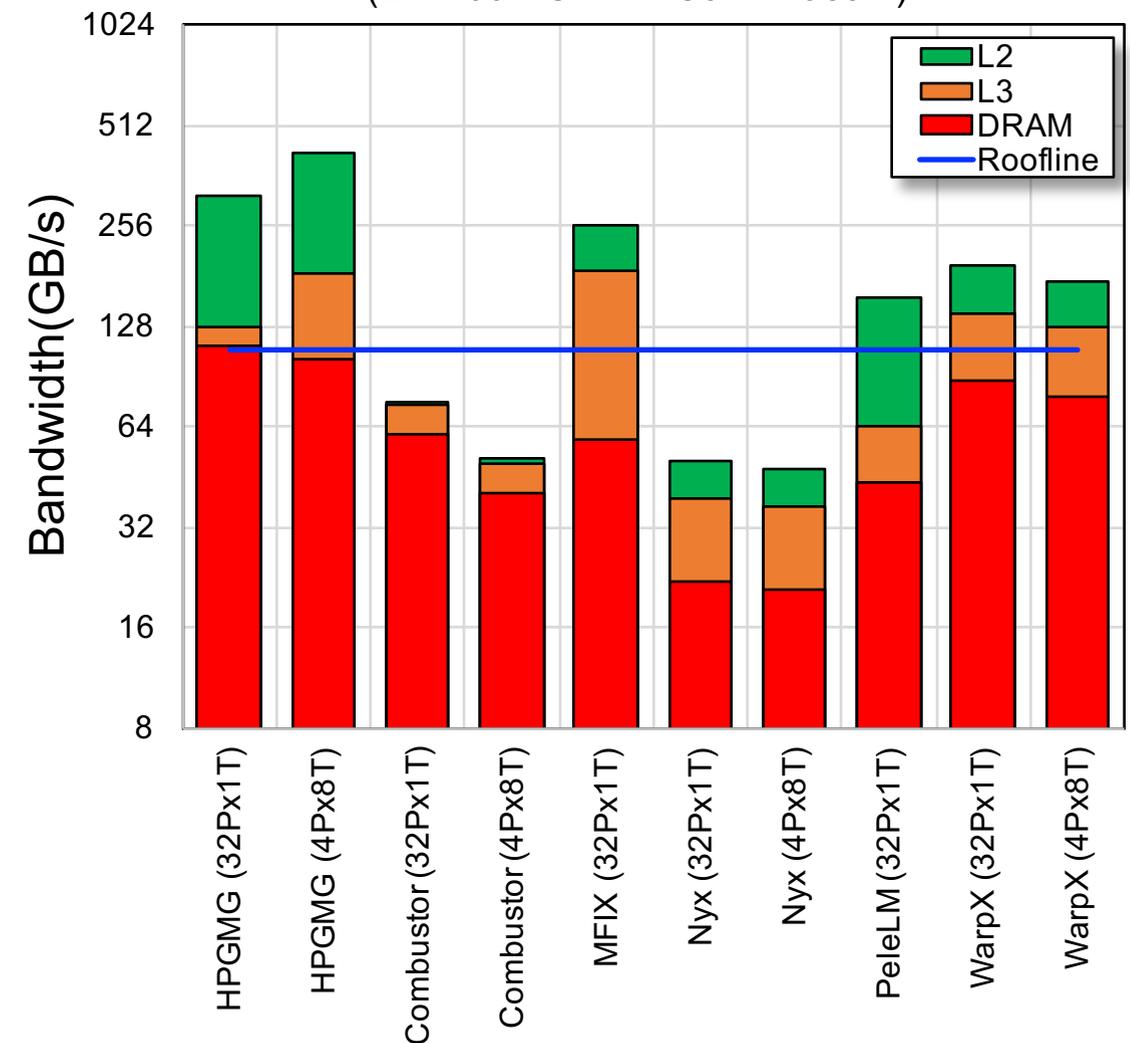


DRAM-only Roofline was insufficient for PICSAR

Evaluation of LIKWID

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

AMReX Application Characterization
(2Px16c HSW == Cori Phase 1)

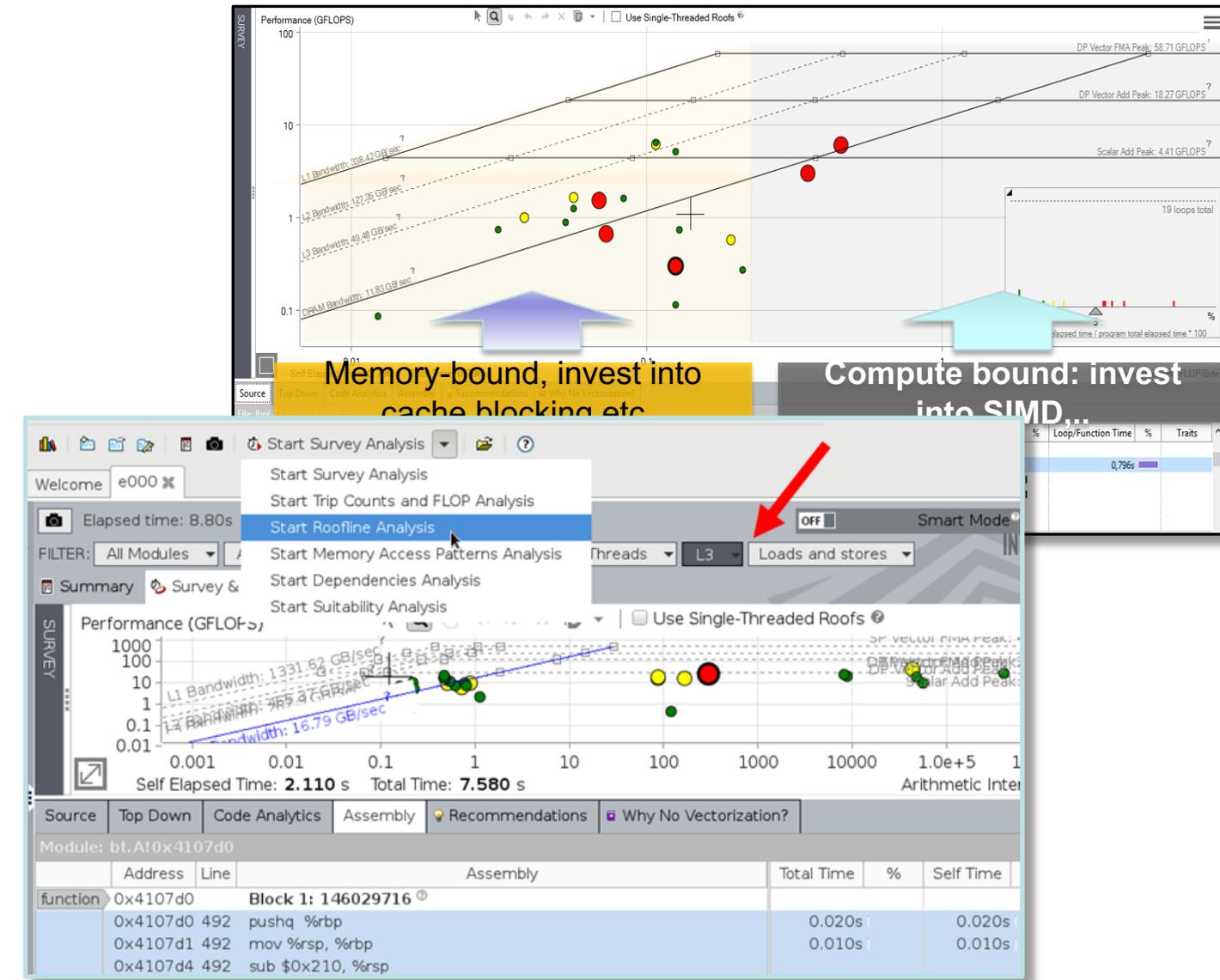


Need an integrated solution...

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

Intel Advisor

- Includes Roofline Automation...
 - ✓ Automatically instruments applications (one dot per loop nest/function)
 - ✓ Computes FLOPS and AI for each function (**CARM**)
 - ✓ AVX-512 support that incorporates masks
 - ✓ **Integrated Cache Simulator¹** (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>

¹Technology Preview, not in official product roadmap so far.

Hierarchical Roofline vs. Cache-Aware Roofline

*...understanding different Roofline
formulations in Advisor*

There are two Major Roofline Formulations:

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

¹Technology 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

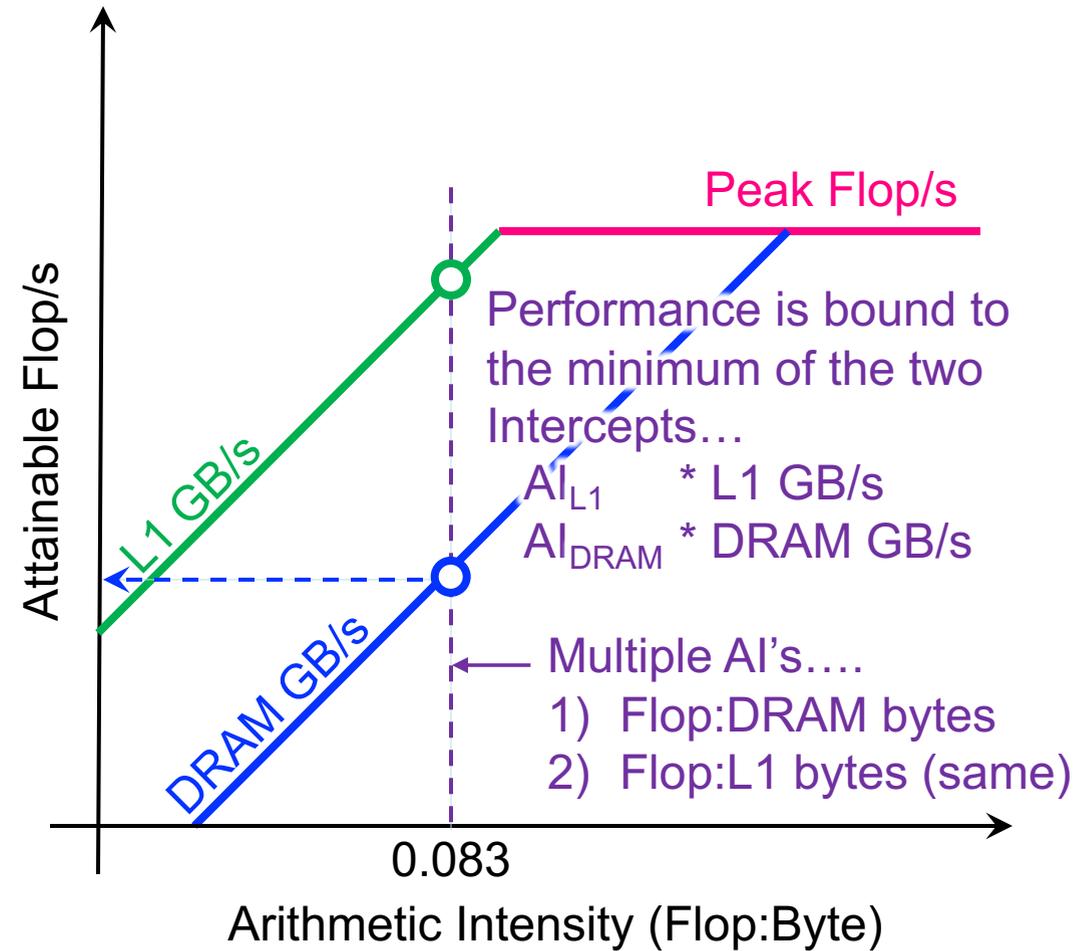
Example: STREAM

- 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 $i+\text{delta}$ for any delta .
- ... leads to a DRAM AI equal to the L1 AI

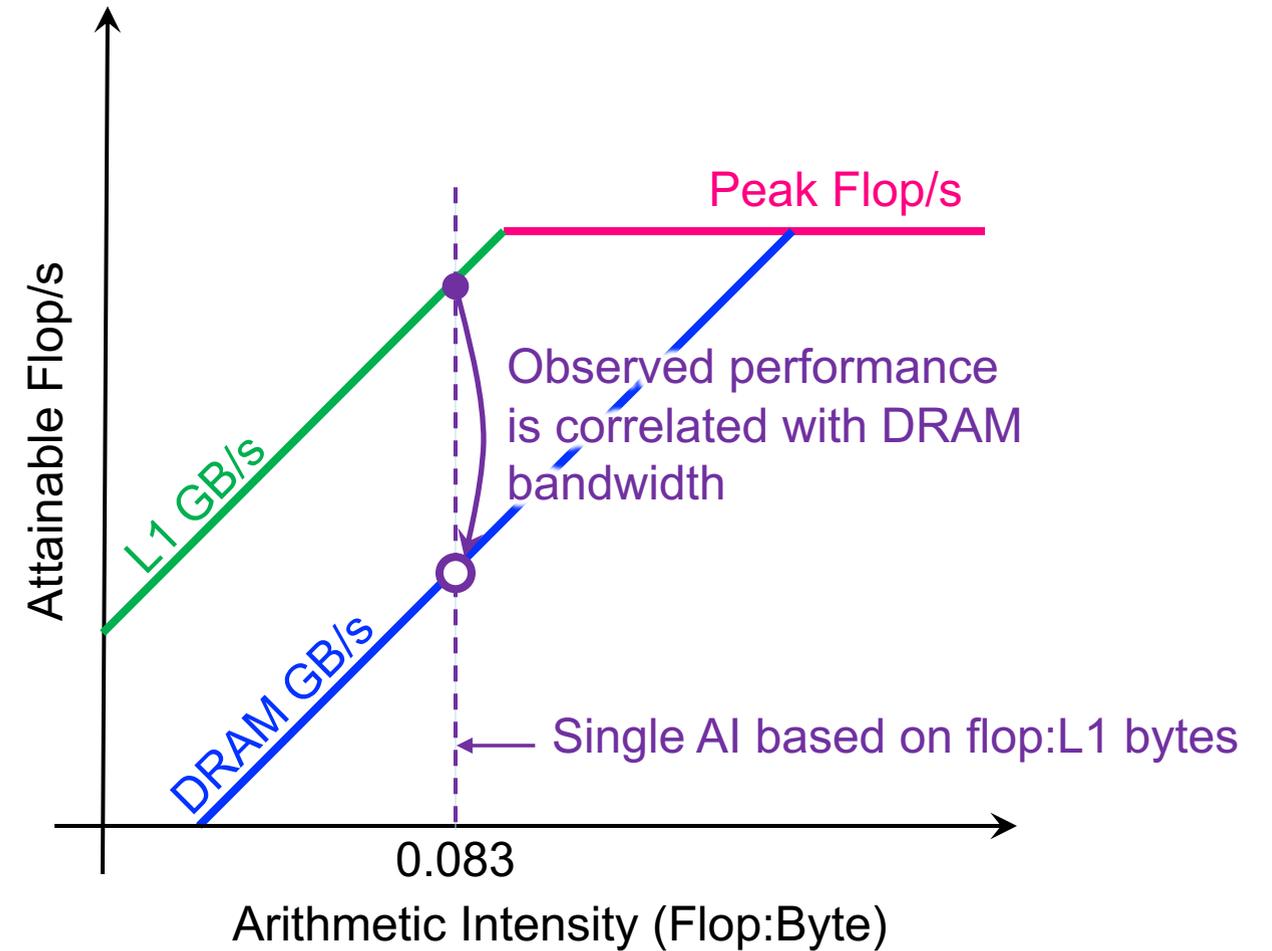
```
#pragma omp parallel for  
for(i=0;i<N;i++){  
    z[i] = x[i] + alpha*y[i];  
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



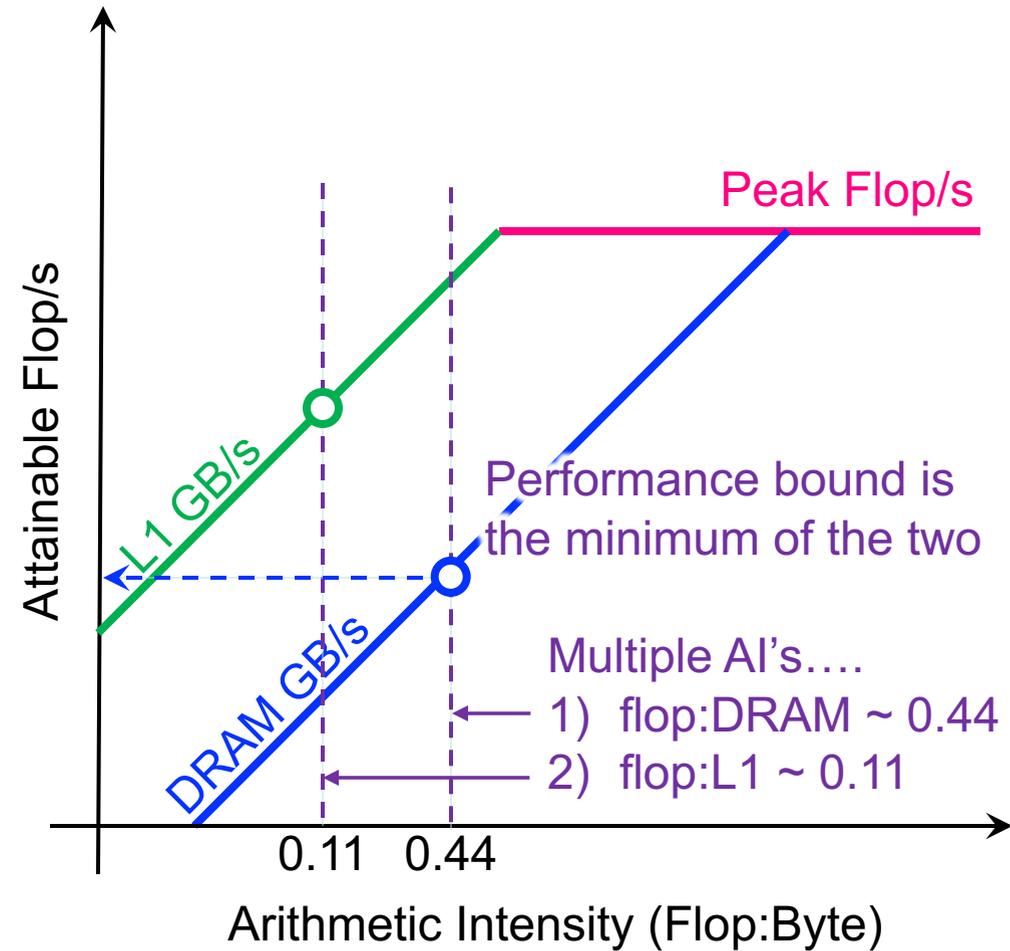
Example: 7-point Stencil (Small Problem)

- L1 AI...
 - 7 flops
 - 7 x 8B load (old)
 - 1 x 8B store (new)
 - = 0.11 flops per byte
 - some compilers may do register shuffles to reduce the number of loads.
- Moderate cache reuse...
 - `old[ijk]` is reused on subsequent iterations of `i,j,k`
 - `old[ijk-1]` is reused on subsequent iterations of `i`.
 - `old[ijk-jStride]` is reused on subsequent iterations of `j`.
 - `old[ijk-kStride]` is reused on subsequent iterations of `k`.
- ... leads to DRAM AI larger than the L1 AI

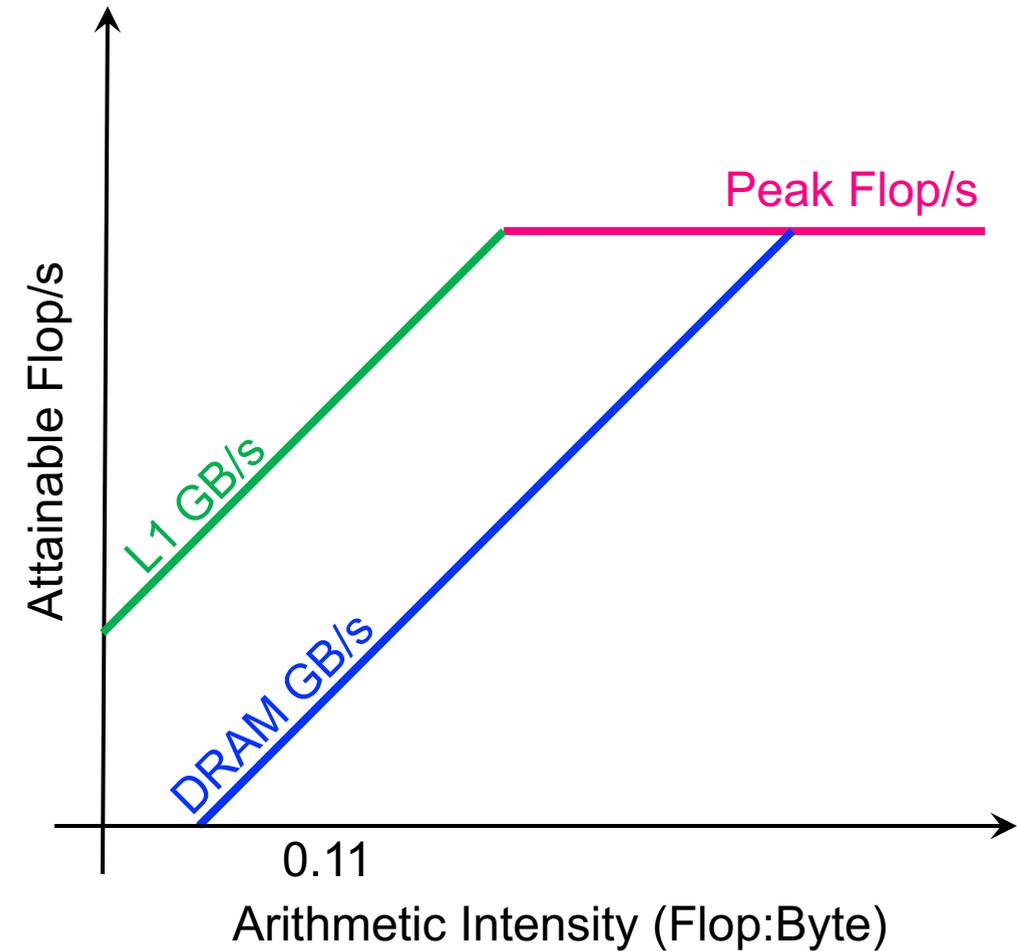
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
    int ijk = i + j*jStride + k*kStride;
    new[ijk] = -6.0*old[ijk
                    + old[ijk-1
                    + old[ijk+1
                    + old[ijk-jStride]
                    + old[ijk+jStride]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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

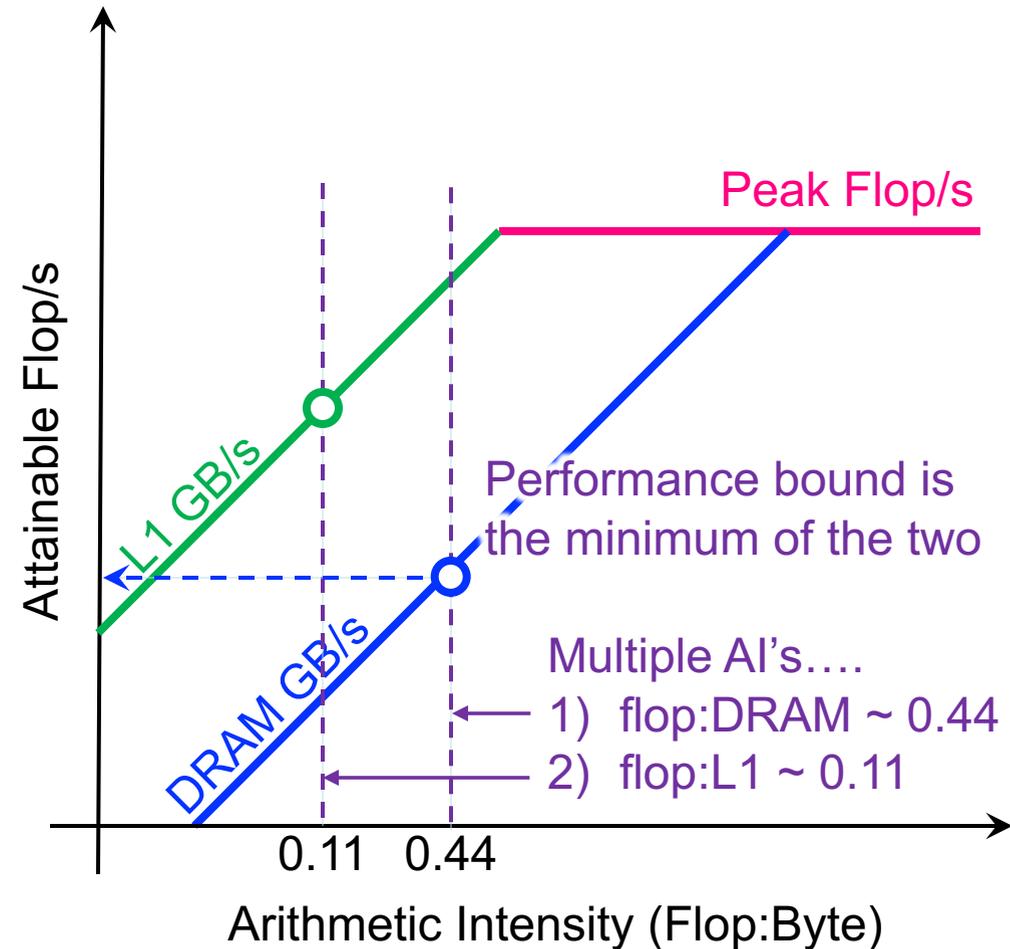


Cache-Aware Roofline

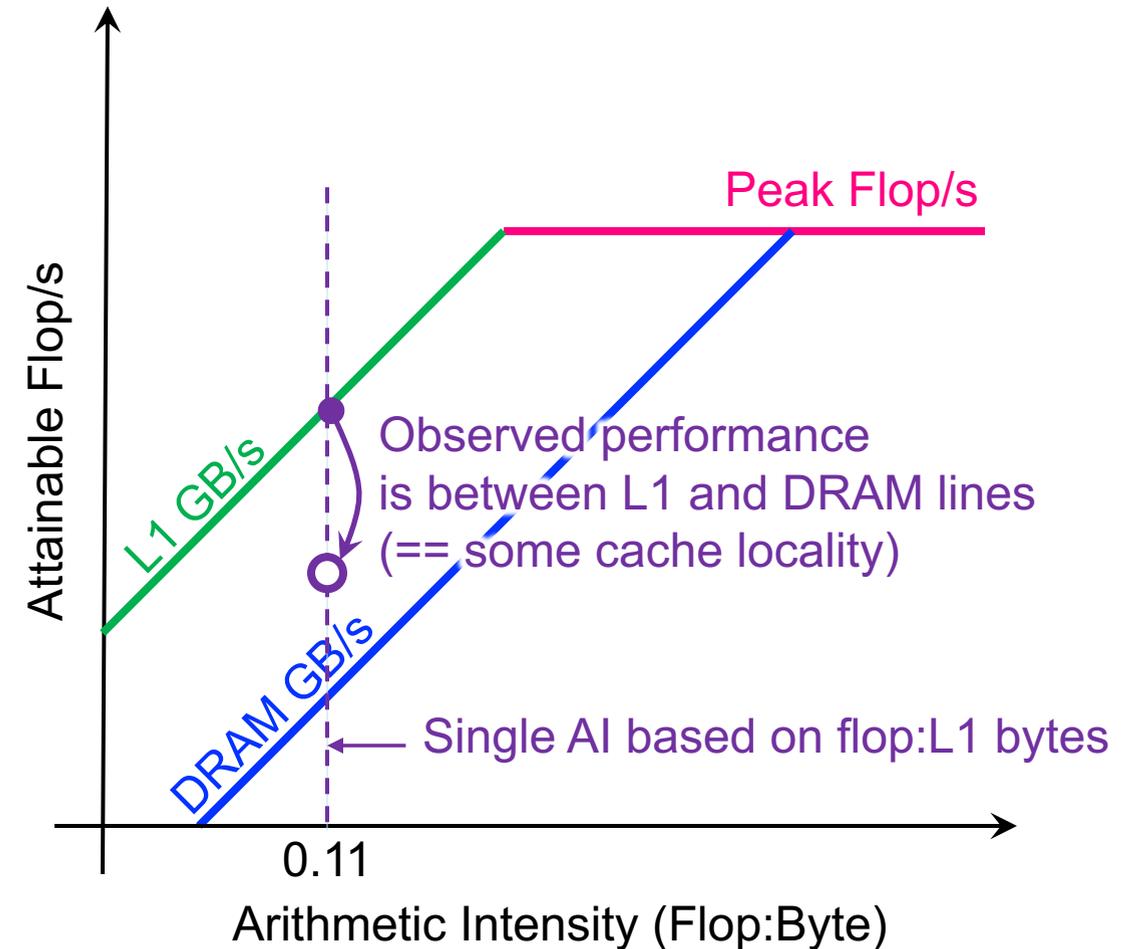


Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

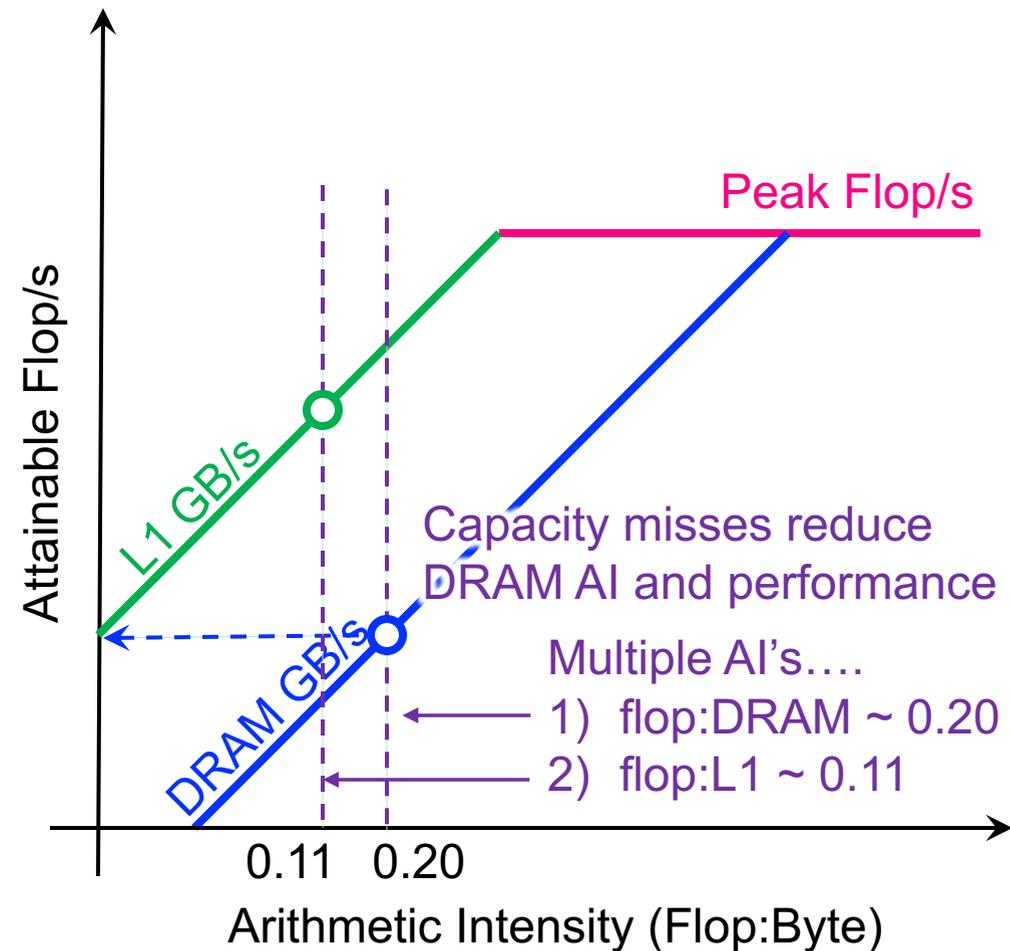


Cache-Aware Roofline

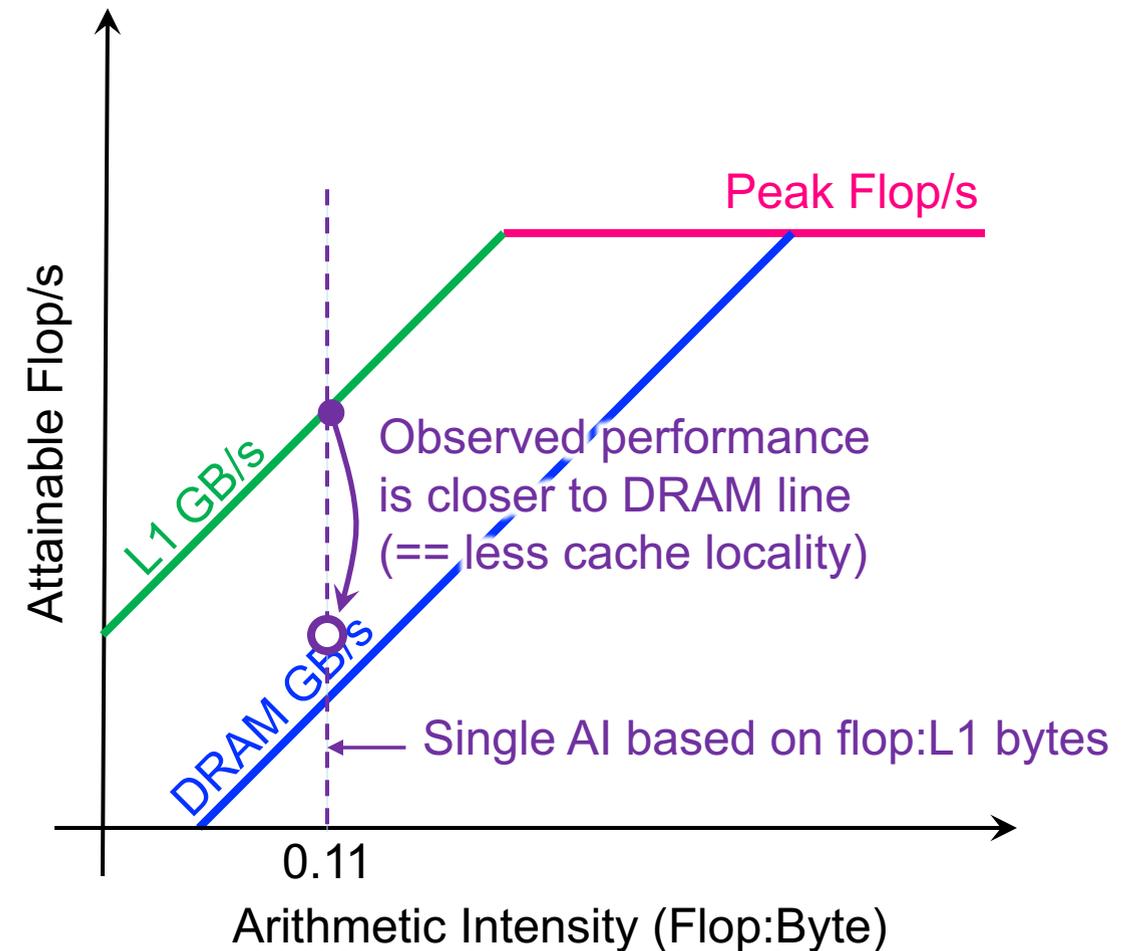


Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

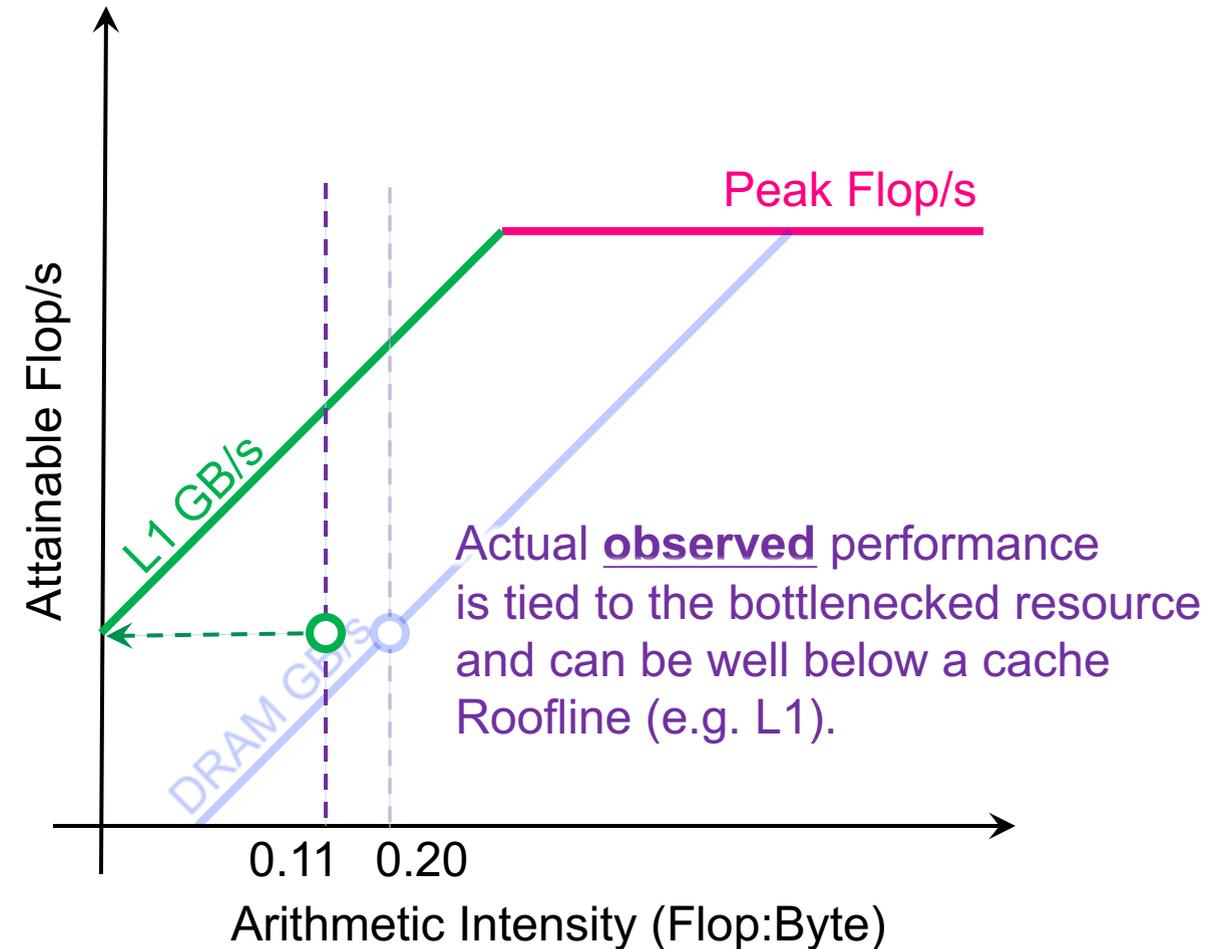


Cache-Aware Roofline

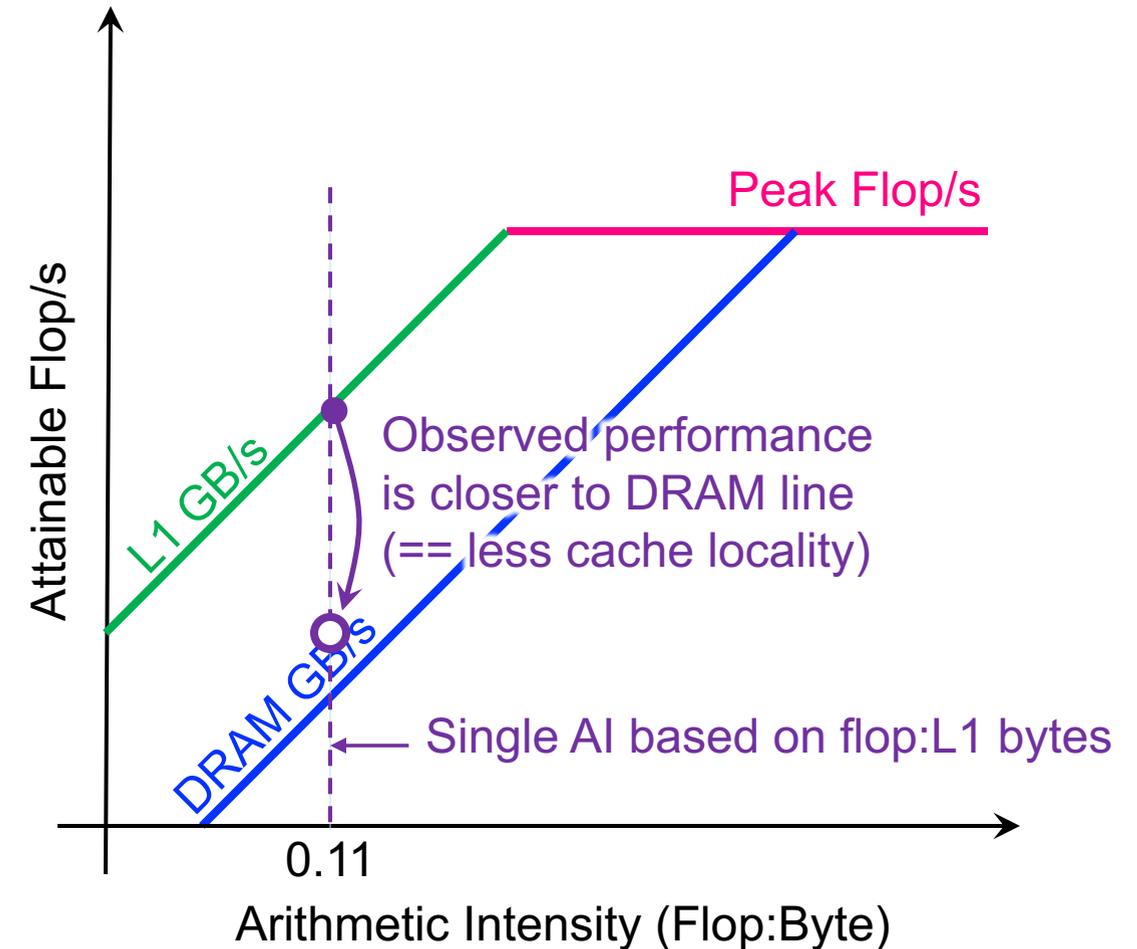


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

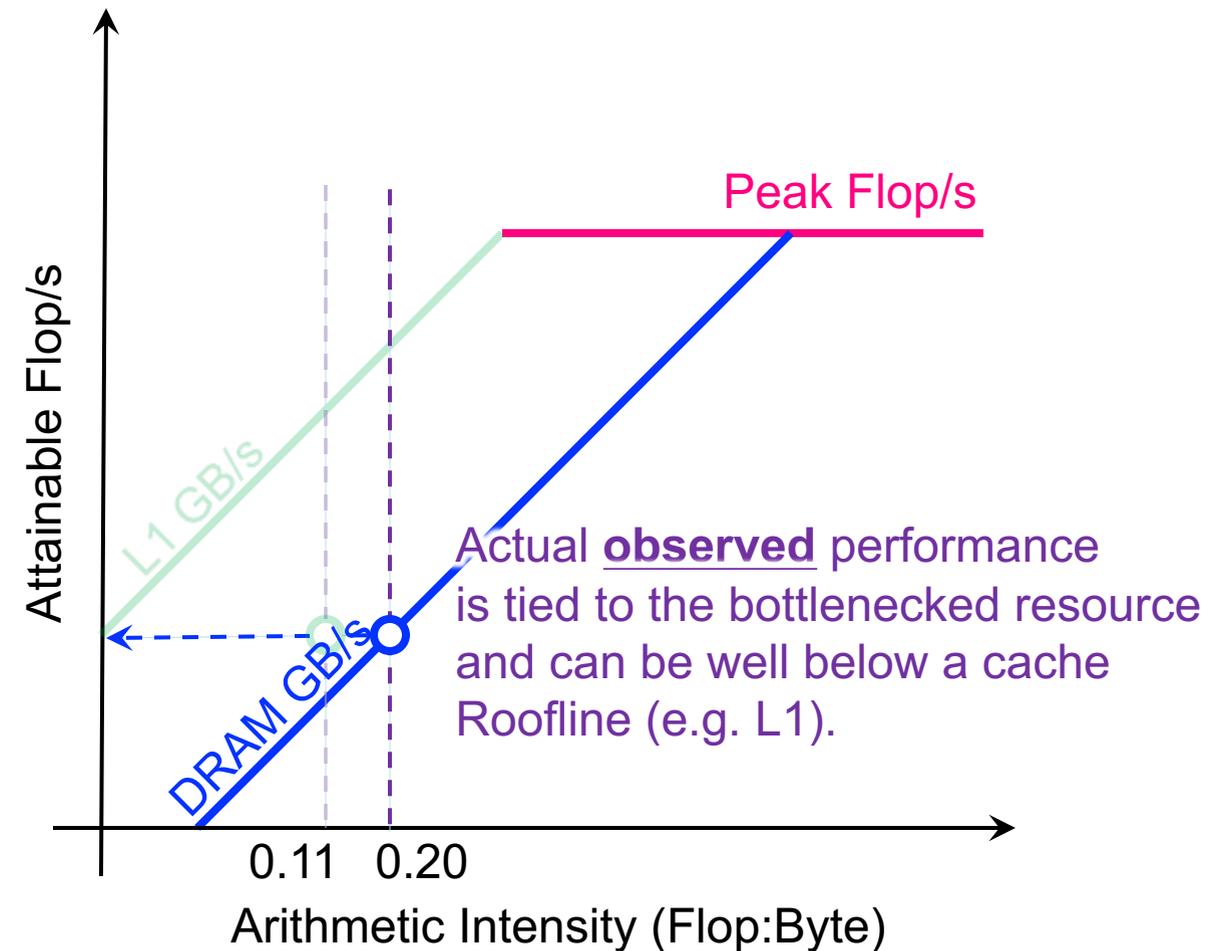


Cache-Aware Roofline

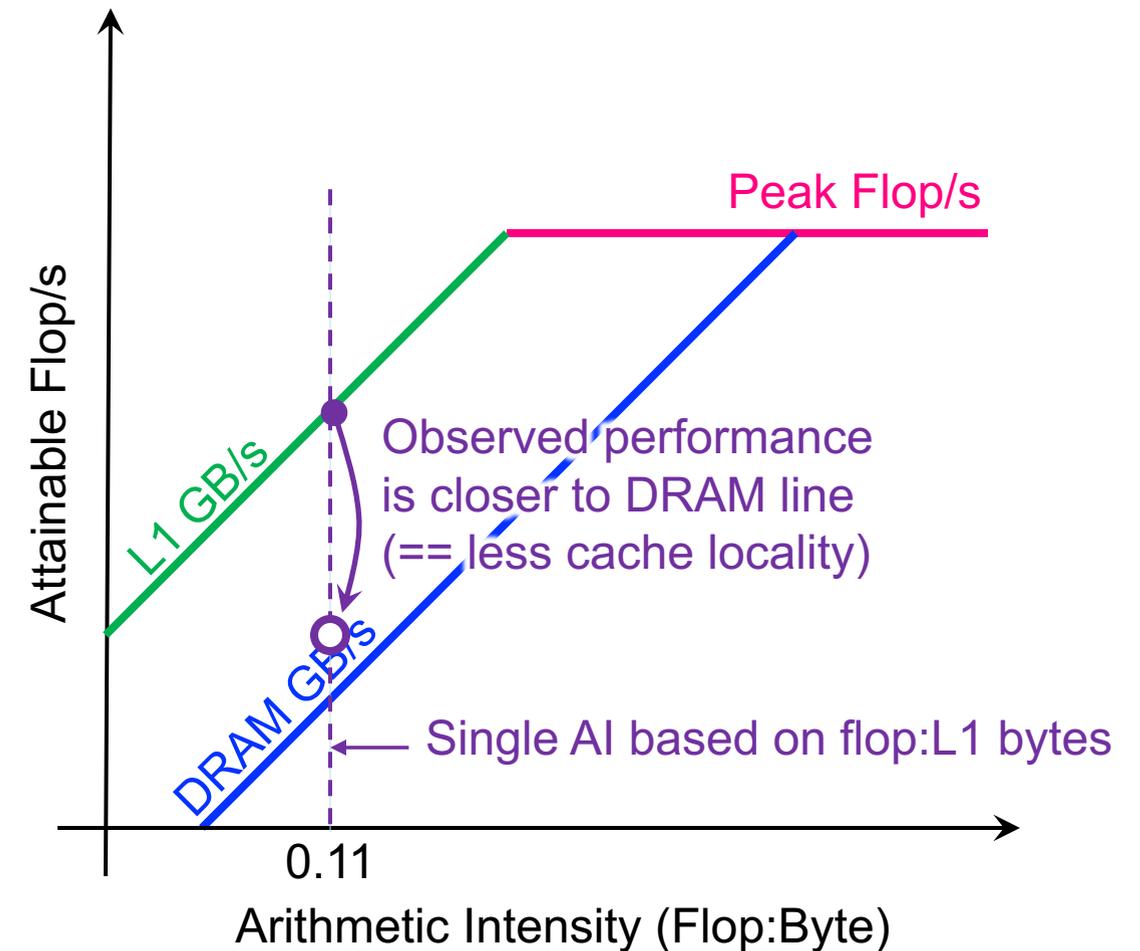


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Cache-Aware Roofline

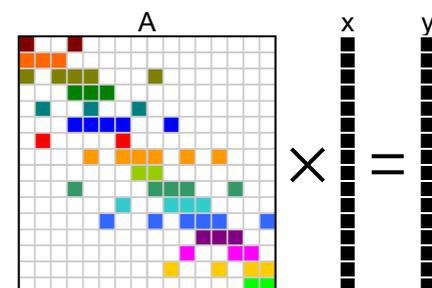




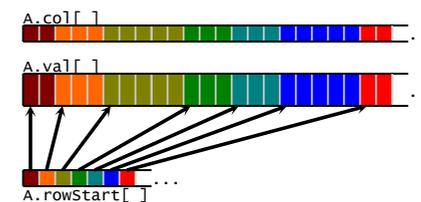
Example of Roofline in Practice

Sparse Matrix Vector Multiplication

- What's a Sparse Matrix ?
 - Most entries are 0.0
 - Performance advantage in only storing/operating on the nonzeros
 - Requires significant meta data to reconstruct the matrix structure
- What's SpMV ?
 - Evaluate $y=Ax$ where A is a sparse matrix, x & y are dense vectors
- Challenges
 - **Very low arithmetic intensity (often <0.166 flops/byte)**
 - Difficult to exploit ILP (bad for pipelined or superscalar),
 - Difficult to exploit DLP (bad for SIMD)



(a)
algebra conceptualization



(b)
CSR data structure

```
for (r=0; r<A.rows; r++) {
  double y0 = 0.0;
  for (i=A.rowStart[r]; i<A.rowStart[r+1]; i++){
    y0 += A.val[i] * x[A.col[i]];
  }
  y[r] = y0;
}
```

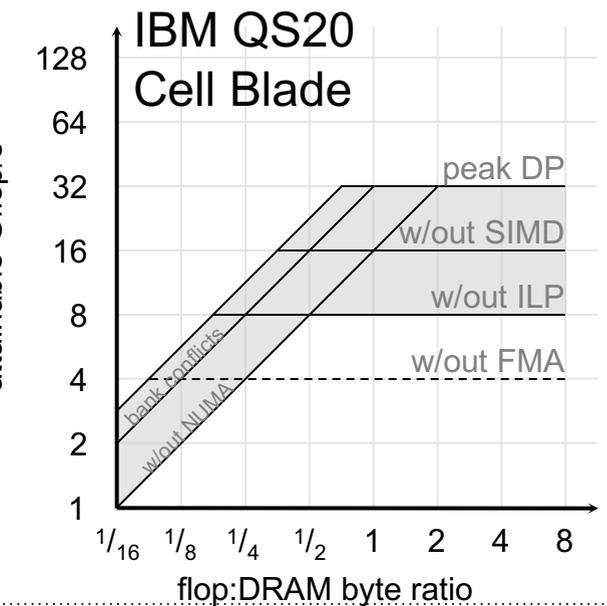
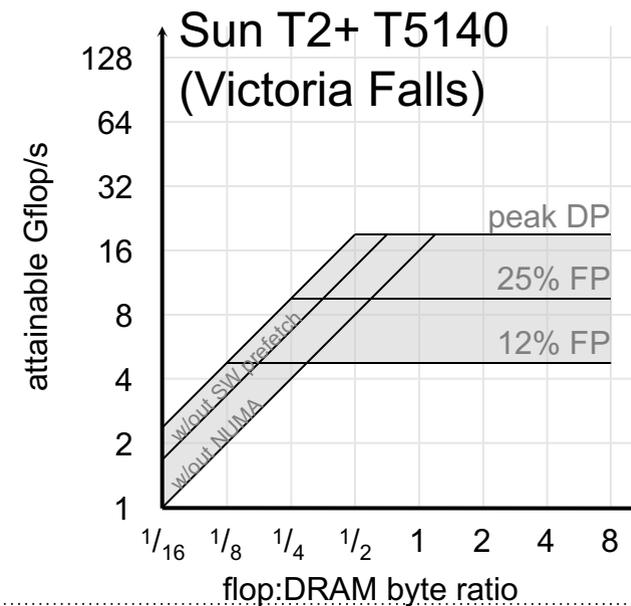
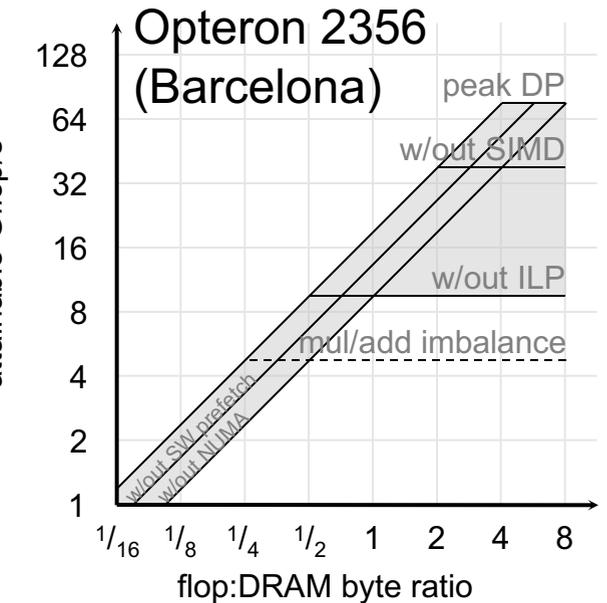
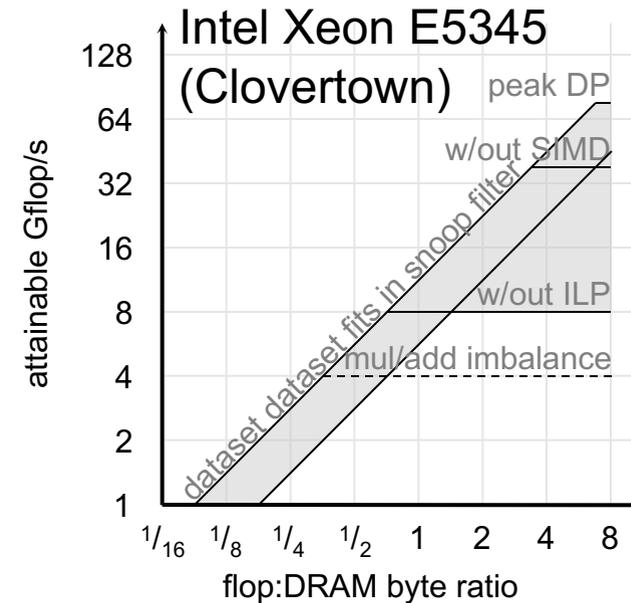
(c)
CSR reference code

Roofline model for SpMV

- Double precision roofline models
- In-core optimizations 1..i
- DRAM optimizations 1..j

$$GFlops_{i,j}(AI) = \min \left\{ \begin{array}{l} \text{InCoreGFlops}_i \\ \text{StreamBW}_j * AI \end{array} \right.$$

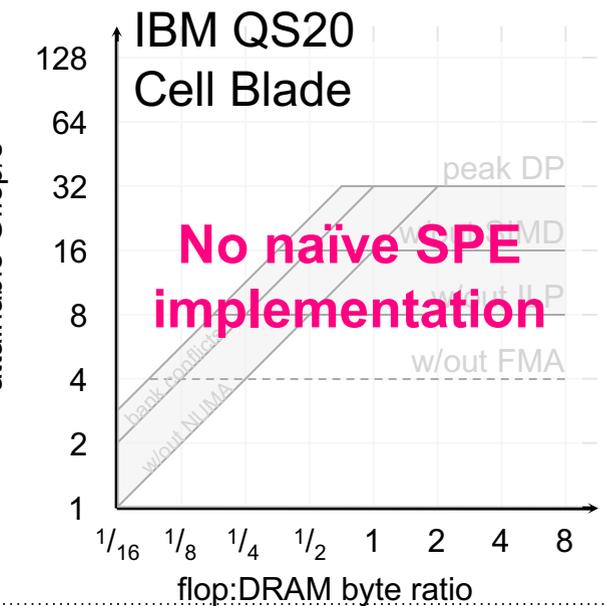
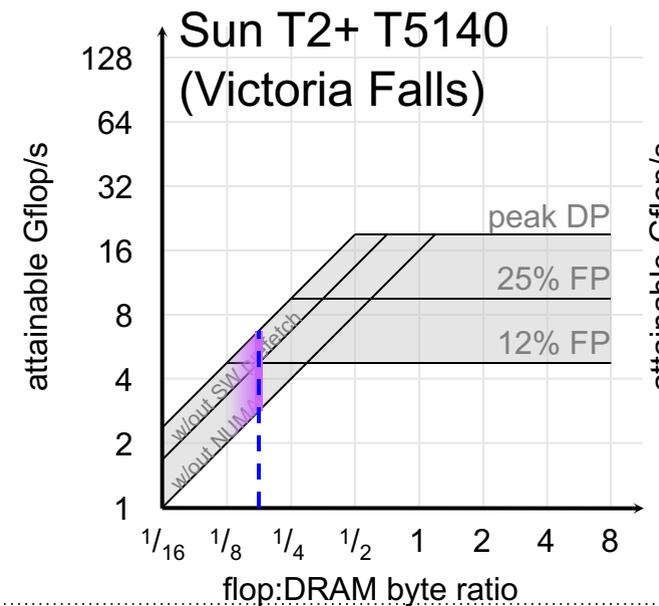
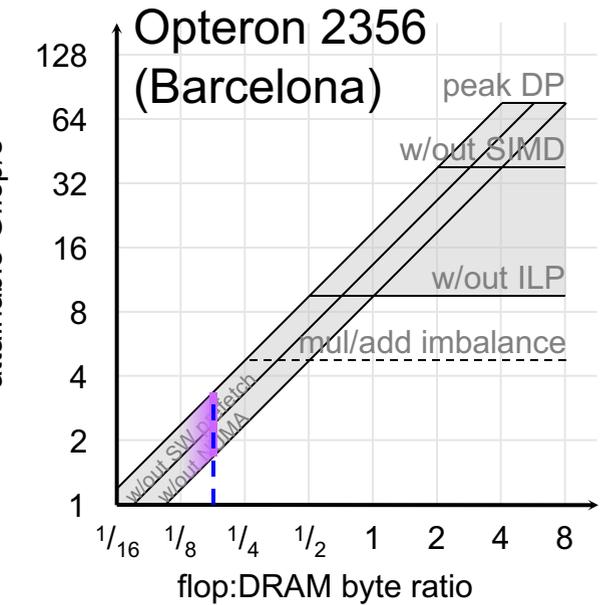
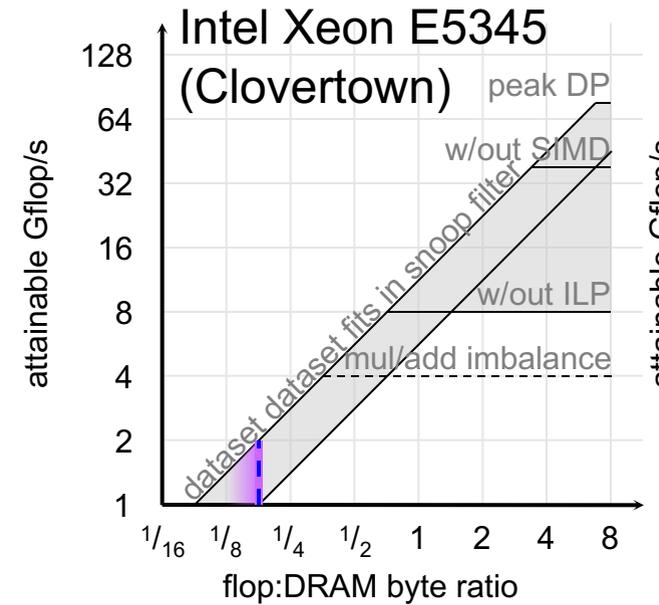
- FMA is inherent in SpMV (place at bottom)



Roofline model for SpMV

(overlay arithmetic intensity)

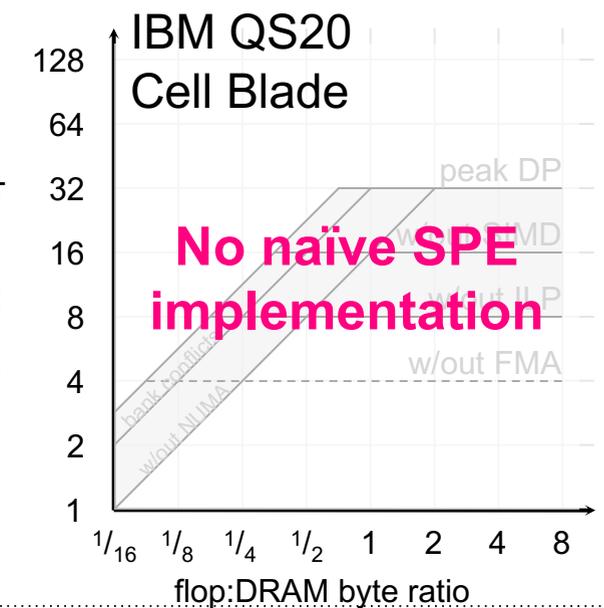
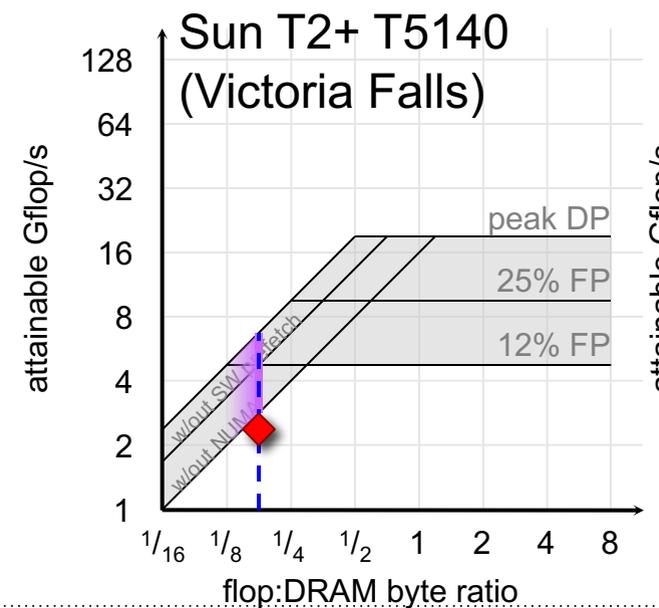
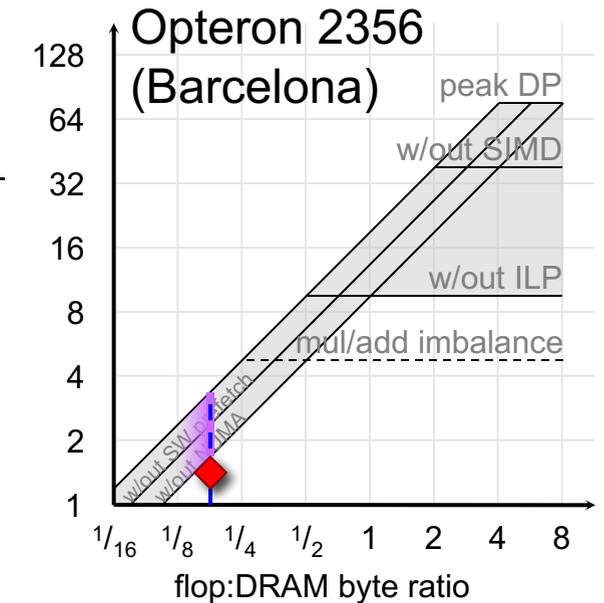
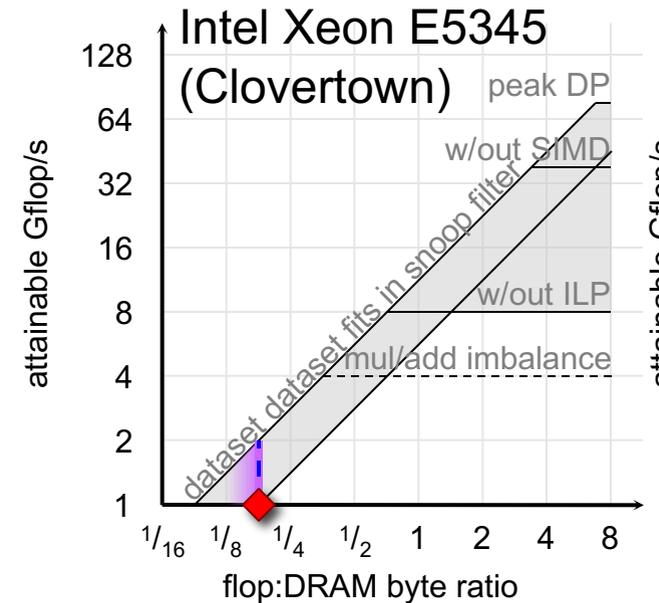
- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166



Roofline model for SpMV

(out-of-the-box parallel)

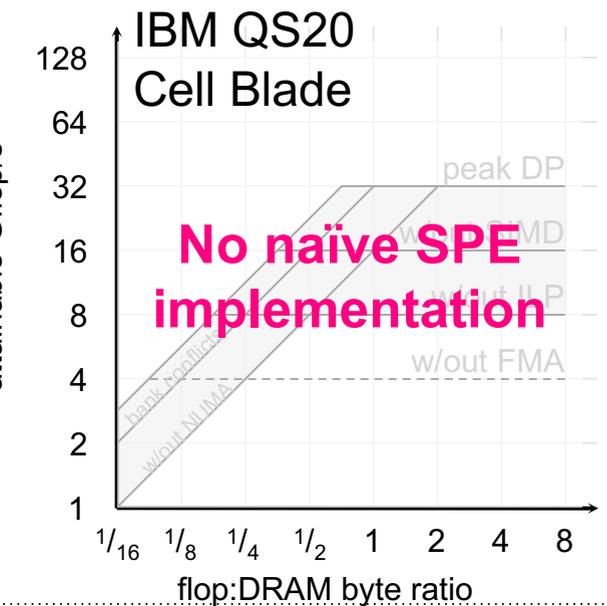
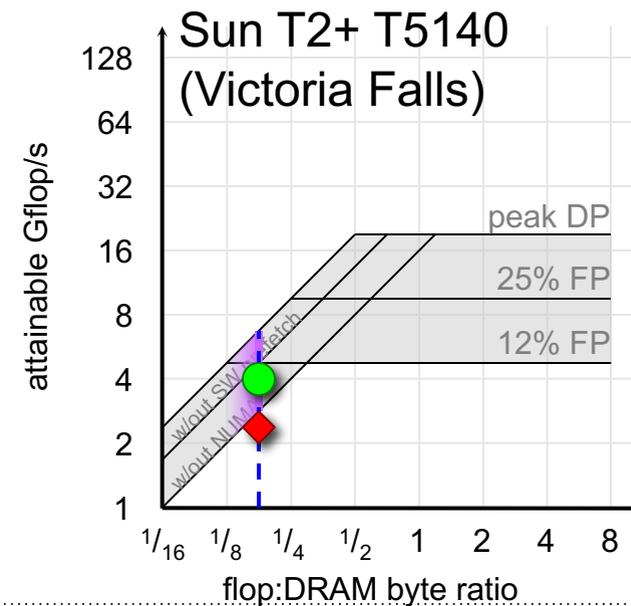
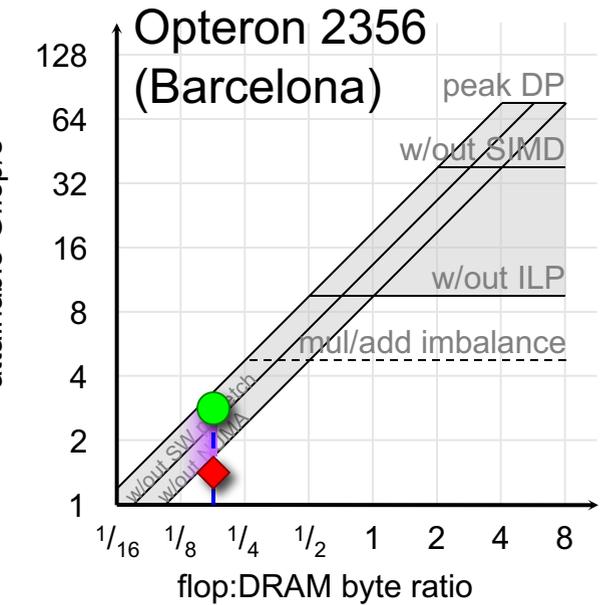
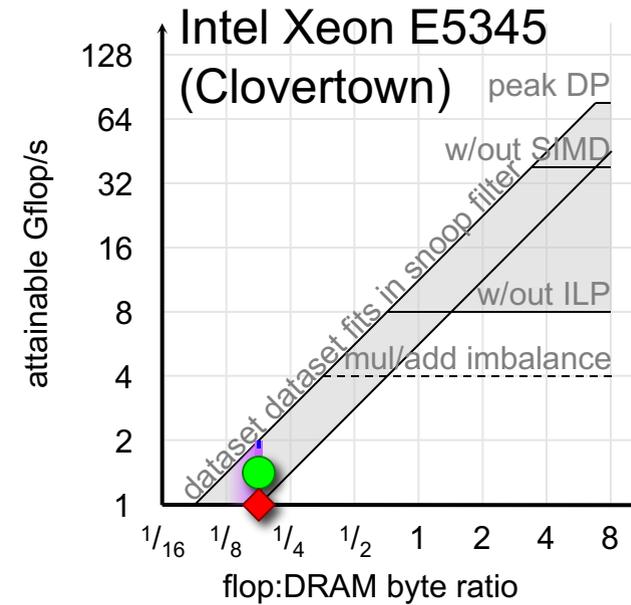
- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166
- For simplicity: dense matrix in sparse format



Roofline model for SpMV

(NUMA & SW prefetch)

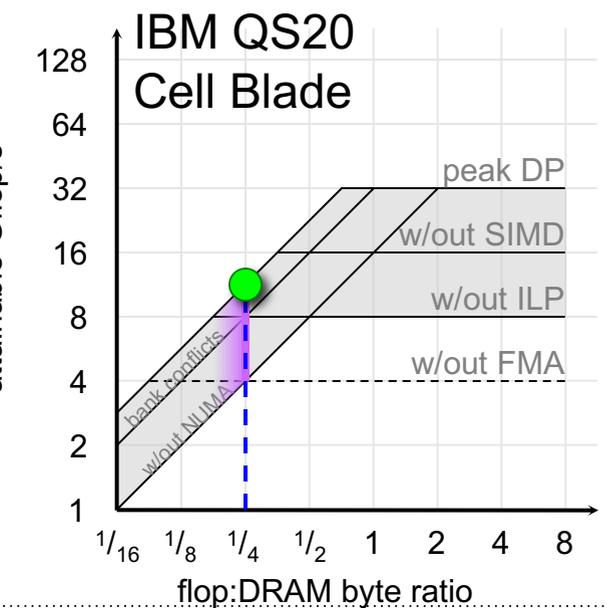
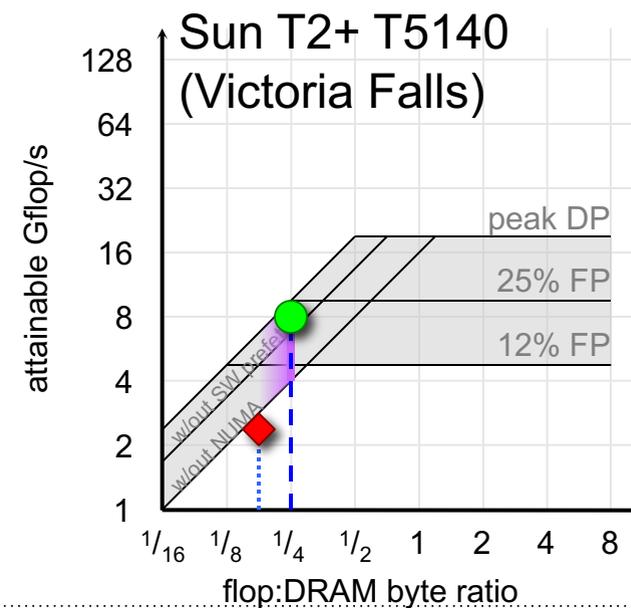
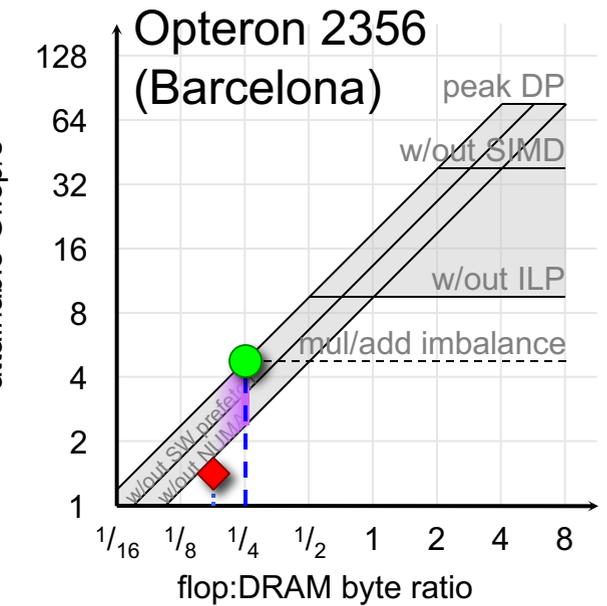
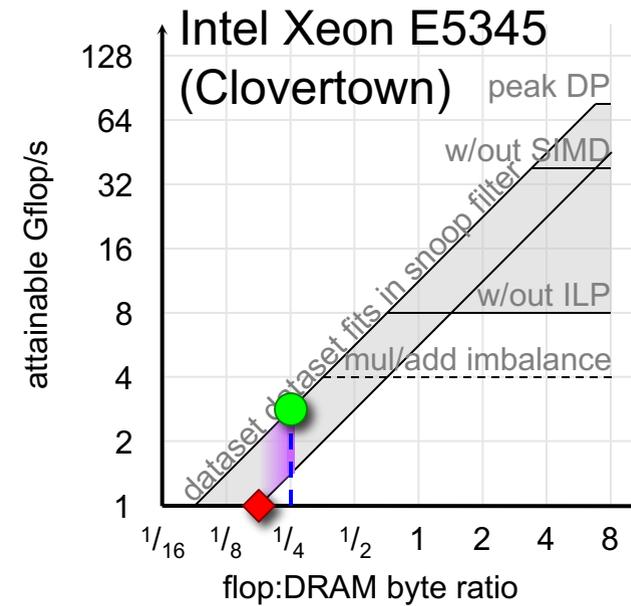
- compulsory flop:byte ~ 0.166
- utilize all memory channels



Roofline model for SpMV

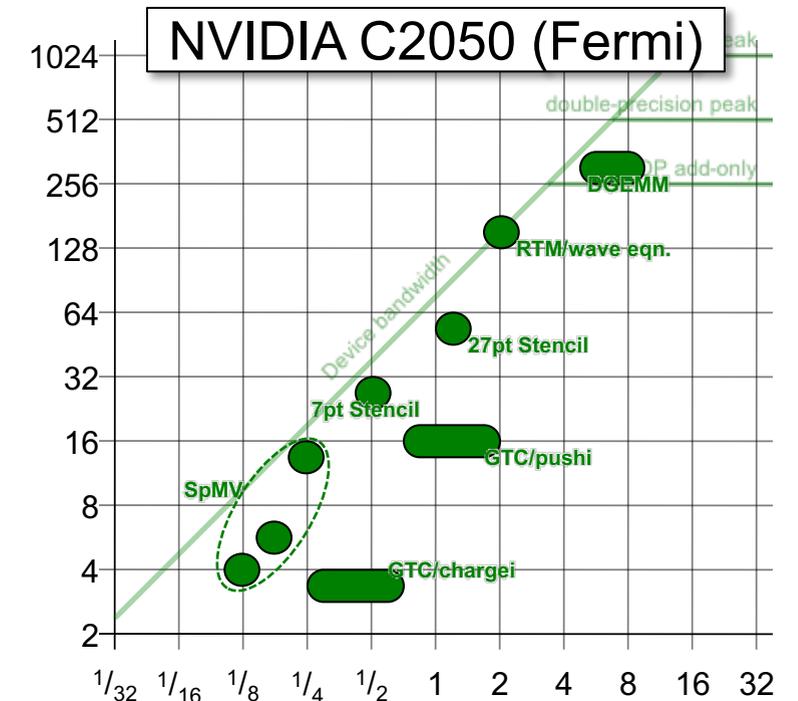
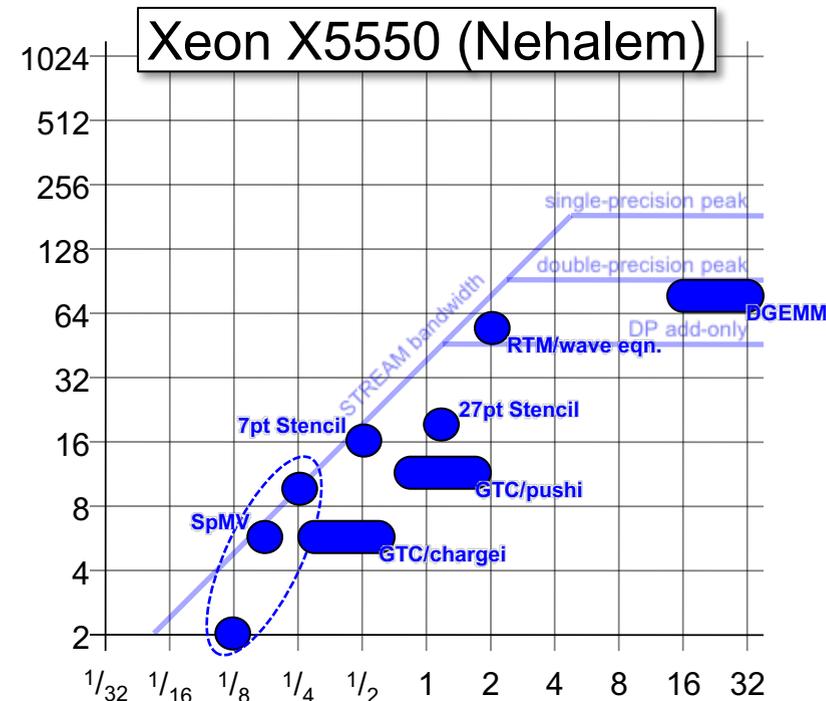
(matrix compression)

- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions



Roofline on CPUs and GPUs

- In 2010, we began to use Roofline to compare CPU and GPU performance for a variety of double-precision kernels
 - Flop/s were theoretical book values;
 - Memory bandwidth from STREAM or SHOC
 - AI was based on compulsory data movement
 - Optimized kernel performance was well-correlated with Roofline for both platforms.
 - Some irregular applications (PIC) underperformed and motivated Further study.





BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY



U.S. DEPARTMENT OF
ENERGY

Questions?



BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY



U.S. DEPARTMENT OF
ENERGY

Backup

Intel Advisor:

Introduction and General Usage

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

***DRAM Roofline and OS/X Advisor GUI:** These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.

Intel Advisor

- **Integrated Performance Analysis Tool**
 - Performance information including timings, flops, and trip counts
 - Vectorization Tips
 - Memory footprint analysis
 - **Originally used the Cache-Aware Roofline Model**
 - All connected back to source code

- **CRD/NERSC began a collaboration with Intel**
 - Ensure Advisor runs on Cori in user-mode
 - Push for **Hierarchical (Integrated) Roofline**
 - Make it functional/scalable to many MPI processes across multiple nodes
 - Validate results on NESAP, SciDAC, and ECP codes

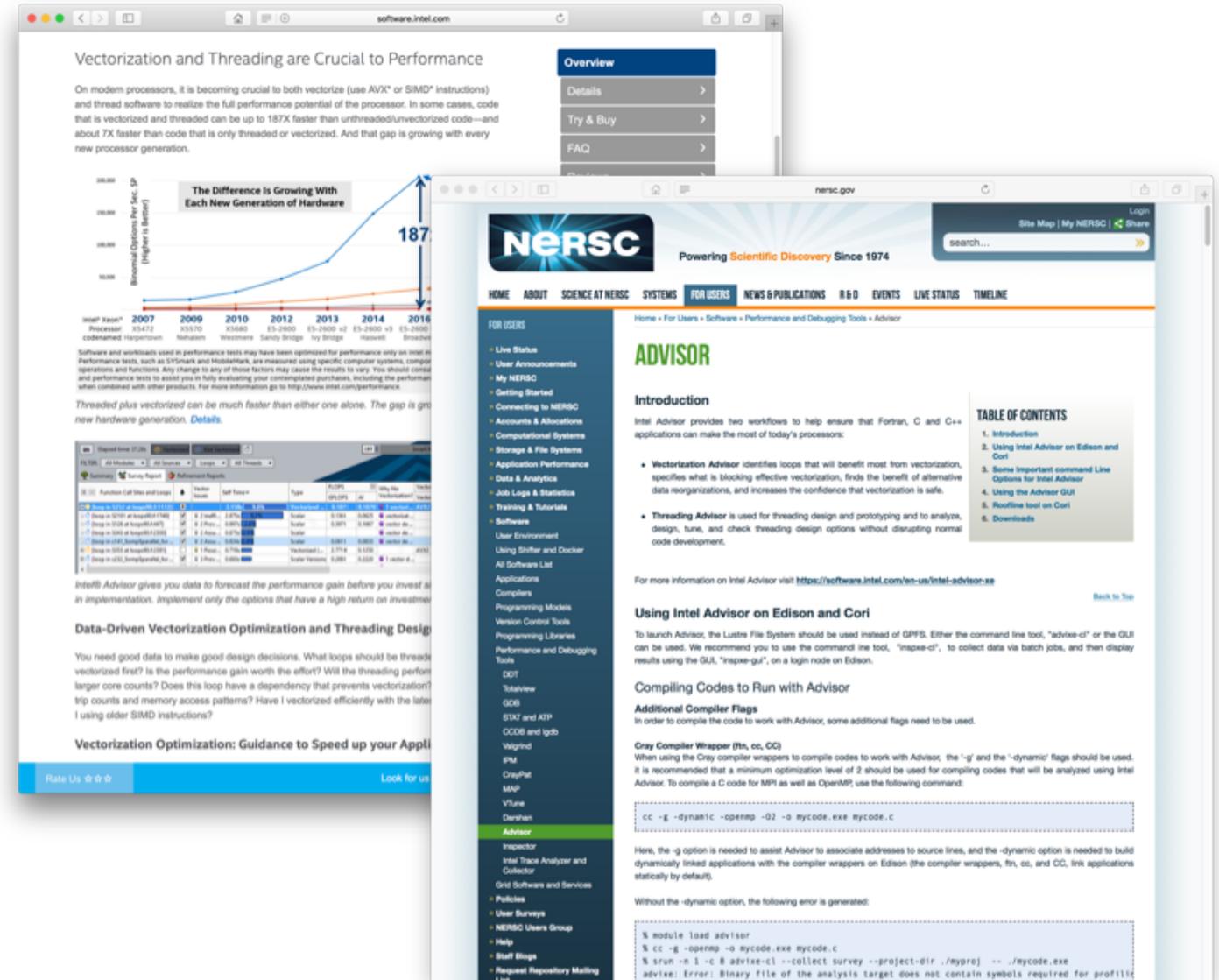
Intel Advisor (Useful Links)

Background

- <https://software.intel.com/en-us/intel-advisor-xe>
- <https://software.intel.com/en-us/articles/getting-started-with-intel-advisor-roofline-feature>
- <https://www.youtube.com/watch?v=h2QEM1HpFgg>

Running Advisor on NERSC Systems

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



Using Intel Advisor at NERSC

■ Compile...

use '-g' when compiling

■ Submit Job...

```
% salloc -perf=vtune
```

-or-

```
#SBATCH -perf=vtune
```

<<< interactive sessions; --perf only needed for DRAM Roofline

<<< batch submissions; --perf only needed for DRAM Roofline

Benchmark...

```
% module load advisor
```

```
% export ADVIXE_EXPERIMENTAL=roofline_ex <<< only needed for DRAM Roofline
```

```
% srun [args] advixe-cl -collect survey -no-stack-stitching -project-dir $DIR -- ./a.out [args]
```

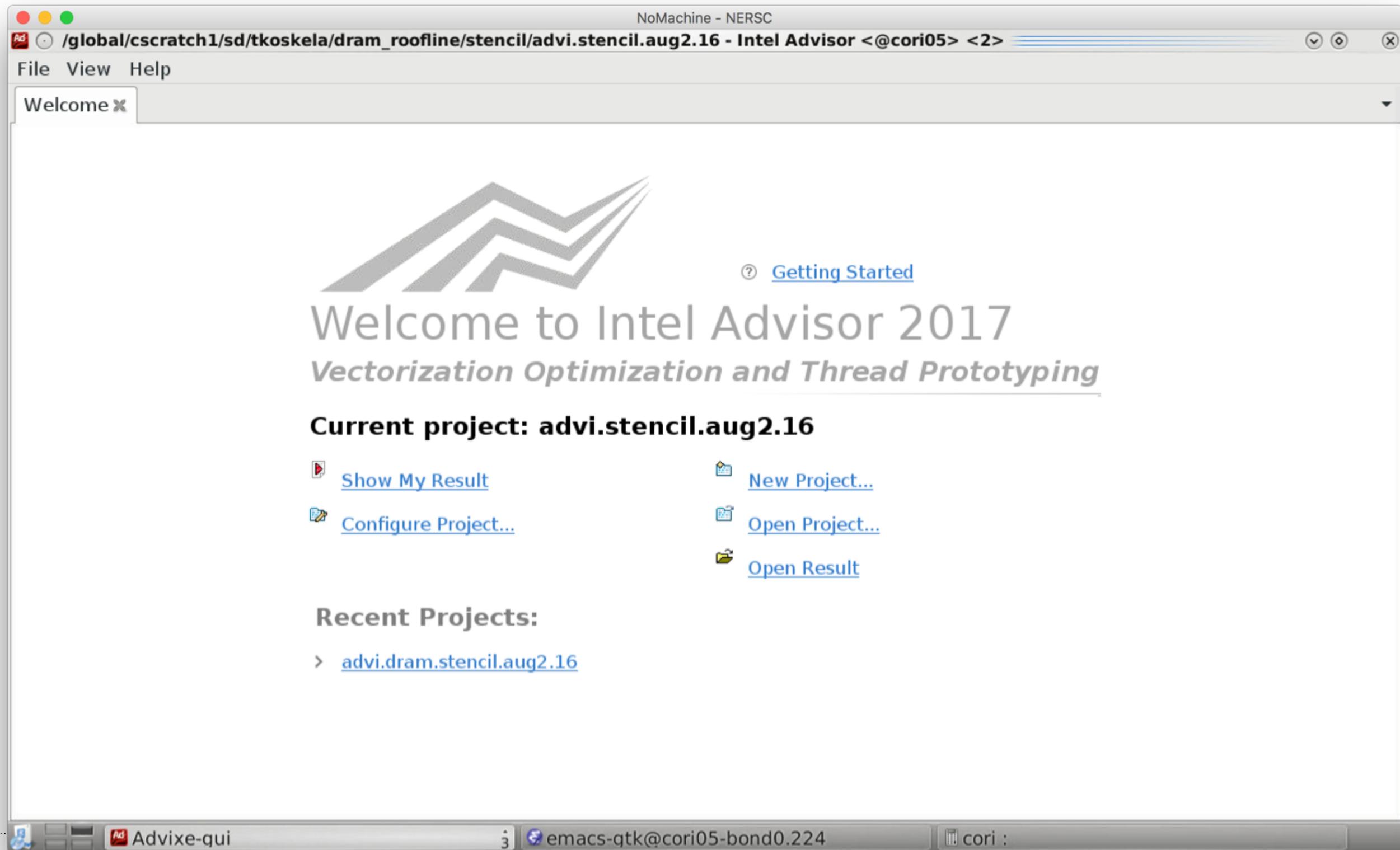
```
% srun [args] advixe-cl -collect tripcounts -flops-and-masks -callstack-flops -project-dir $DIR -- ./a.out [args]
```

■ Use Advisor GUI...

```
% module load advisor
```

```
% export ADVIXE_EXPERIMENTAL=roofline_ex <<< only needed for DRAM Roofline
```

```
% advixe-gui $DIR
```



NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized

FILTER: All Modules All Sources

Summary Survey & Roofline Refinement Reports

Program metrics

Elapsed Time 50.50s

Vector Instruction Set AVX Number of CPU Threads 16

Total GFLOP Count 753.95 Total GFLOPS 14.93

Total Arithmetic Intensity[®] 0.12

Loop metrics

Total CPU time 806.22s 100.0%

Time in 5 vectorized loops 641.62s 79.6%

Time in scalar code 164.60s 20.4%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency[®] 3.81x 95%

Program Approximate Gain[®] 3.23x

Top time-consuming loops[®]

Loop	Self Time [®]	Total Time [®]	Trip Counts [®]
[loop in bench_stencil_ver2\$omp\$parallel_for@102 at stencil_v2.c:108]	160.035s	160.035s	31; 3; 2; 3
[loop in bench_stencil_ver3\$omp\$parallel_for@146 at stencil_v2.c:152]	159.953s	159.953s	32; 2
[loop in bench_stencil_ver4\$omp\$parallel_for@193 at stencil_v2.c:201]	159.595s	159.595s	130
[loop in bench_stencil_ver1\$omp\$parallel_for@62 at stencil_v2.c:65]	159.307s	159.307s	31; 3; 2; 3

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	
					GFLOPS	AI
[loop in bench_stencil_ver4\$...		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3\$...	1 Ineffective peeled/...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
[loop in bench_stencil_ver2\$...	1 Ineffective peeled/...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
[loop in bench_stencil_ver1\$...	1 Ineffective peeled/...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
[loop in bench_stencil_ver0\$...	1 Potential underutil...	157.994s	157.994s	Scalar	9.009	0.117

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:108 bench_stencil_ver2\$omp\$parallel_for@102

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
103	for (tile=0; tile<jTiles*kTiles; tile++){	0.010s				
104	int kLo = 16*(tile/jTiles);					
105	int jLo = 16*(tile%jTiles);					
106	for (k=kLo; k<kLo+16; k++){	0.008s				
107	for (j=jLo; j<jLo+16; j++){	0.092s				
108	for (i=0; i<dim; i++){	1.500s		160.035s		
109	int ijk = i + j*jStride + k*kStride;					
110	new[ijk] = -6.0*old[ijk]	26.216s				FMA
Selected (Total Time):		1.500s				

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	
					GFLOPS	AI
[loop in bench_stencil_ver4 ...]		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3 ...]	1 Ineffective peeled/ ...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
[loop in bench_stencil_ver2 ...]	1 Ineffective peeled/ ...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
[loop in bench_stencil_ver1 ...]	1 Ineffective peeled/ ...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
[loop in bench_stencil_v ...]	1 Potential under ...	157.994s	157.994s	Scalar	9.009	0.117

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Type	FLOPS
					GFLOPS
_INTERNAL_26_____src_z_Linux_util_cpp_2d702c13::[OpenM	93.8%	755.843s	0.000s	Function	0.998
[loop in _INTERNAL_26_____src_z_	93.8%	755.843s	0.000s	Function	0.998
INTERNAL_26_____src_z_Linux_util_cpp_2d702c13::[OpenMP worker]	93.8%	755.843s	0.000s	Function	0.998
_kmp_launch_thread	93.8%	755.843s	0.000s	Function	0.998
[loop in _kmp_launch_thread at kmp_runtime.cpp:565	93.8%	755.843s	0.000s	Scalar	0.998
[OpenMP dispatcher]	93.3%	752.000s	0.000s	Function	1.003
bench_stencil_ver2\$omp\$parallel_for@102	18.7%	150.903s	0.120s	Function	1.083
bench_stencil_ver3\$omp\$parallel_for@146	18.7%	150.471s	0.000s	Function	1.097
bench_stencil_ver4\$omp\$parallel_for@193	18.6%	149.989s	0.000s	Function	1.540
bench_stencil_ver1\$omp\$parallel_for@62	18.6%	149.742s	0.000s	Function	0.606

Advixe-qui emacs-gtk@cori05-bond0.224 cori :

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized OFF Smart Mode[®]

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

ROOFLINE	Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
	[loop in bench_stencil_ver4 ...]		159.595s	159.595s	Vectorized (Body)	23.083	0.117
	[loop in bench_stencil_ver3 ...]	1 Ineffective peeled/ ...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
	[loop in bench_stencil_ver2 ...]	1 Ineffective peeled/ ...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
	[loop in bench_stencil_ver1 ...]	1 Ineffective peeled/ ...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
	[loop in bench_stencil_v...	1 Potential under ...	157.994s	157.994s	Scalar	9.009	0.117

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

Loop in
bench_stencil_ver0\$omp\$parallel_for...
at *stencil_v2.c:29*

157.994s
Scalar Total time

157.994s
Self time

Average Trip Counts: 512

Instruction Mix[®]

Memory: 5 Compute: 9 Mixed[®]: 4
Other: 4 Number of Vector Registers: 9

GFLOPS: 9.00873

Code Optimizations

Compiler: Intel(R) C Intel(R) 64
Compiler for applications running on Intel(R) 64,
Version: 17.0.2.174 Build 20170213

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

ROOFLINE	Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
	[loop in bench_stencil_ver4 ...]		159.595s	159.595s	Vectorized (Body)	23.083	0.117
	[loop in bench_stencil_ver3 ...]	1 Ineffective peeled/...	159.953s	159.953s	Vectorized (Body; Re...)	16.274	0.117
	[loop in bench_stencil_ver2 ...]	1 Ineffective peeled/...	160.035s	160.035s	Vectorized (Body; Peel...)	15.662	0.117
	[loop in bench_stencil_ver1 ...]	1 Ineffective peeled/...	159.307s	159.307s	Vectorized (Body; Peel...)	10.218	0.117
	[loop in bench_stencil_v ...]	1 Potential under...	157.994s	157.994s	Scalar	9.009	0.117

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

Module: a.out!0x401690

	Address	Lin.	Assembly	Total Time	%	Self Time	%	Traits
body	0x401690		Block 1:					
	0x401690	34	vmovsdq (%rbp,%rdx,8), %xmm1	0.996s		0.996s		
	0x401696	31	leal (%r13,%r12,1), %r11d	1.728s		1.728s		
	0x40169b	31	movsxd %r11d, %r11	1.008s		1.008s		
	0x40169e	29	inc %r10d	1.794s		1.794s		
	0x4016a1	36	vmovsdq (%rbp,%rdi,8), %xmm2	0.944s		0.944s		
	0x4016a7	29	add %r8, %rdx	49.773s		49.773s		
	0x4016aa	29	add %r8, %rdi	1.456s		1.456s		
Selected (Total Time):				0s				

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized Smart Mode[®]

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Performance (GFLOPS)

Use Single-Threaded Roofs Show Hierarchical Data

L1 Bandwidth: 5516.07 GB/sec?
 L2 Bandwidth: 1801.27 GB/sec?
 L3 Bandwidth: 502.16 GB/sec?
 DRAM Bandwidth: 128.85 GB/sec?

DP Vector FMA Peak: 843.06 GFLOPS?
 DP Vector Add Peak: 210.96 GFLOPS?
 Scalar Add Peak: 57.44 GFLOPS?

Performance: 116.2 GFLOPS
 Arithmetic Intensity: 0.16 FLOP/Byte

Self Elapsed Time: 10.004 s Total Time: 159.595 s Arithmetic Intensity (FLOP/Byte)

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:201 bench_stencil_ver4\$omp\$parallel_for@193

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
200	#pragma vector nontemporal					
201	for(i=0; i<jStride; i++){	3.636s		159.595s		
Selected (Total Time):		3.636s				

Advixe-qui emacs-gtk@cori05-bond0.224 cori :

Intel Advisor: Stencil Roofline Demo*

**DRAM Roofline and OS/X Advisor GUI:* These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.

7-point, Constant-Coefficient Stencil

- Apply to a 512^3 domain on a single NUMA node (single HSW socket)
- Create 5 code variants to highlight effects (as seen in advisor)

```
ver0.      Baseline: thread over outer loop (k), but prevent vectorization
           #pragma novector                                // prevent simd
           int ijk = i*istride + j*jStride + k*kStride; // variable istride to confuse the compiler

ver1.      Enable vectorization
           int ijk = i + j*jStride + k*kStride;           // unit-stride inner loop

ver2.      Eliminate capacity misses
           2D tiling of j-k iteration space              // working set had been O(6MB) per thread

ver3.      Improve vectorization
           Provide aligned pointers and strides

ver4.      Force vectorization / cache bypass
           __assume(jstride%8 == 0);                       // stride by variable is still aligned
           #pragma omp simd, vector nontemporal           // force simd; force cache bypass
```

NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

File View Help

Welcome e000 (read-only) x

Elapsed time: 50.50s Vectorized Not Vectorized OFF Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Function Call Sites and Loops	Self Time	T. Ti.	T.	FLOPS		W N	Vectorized Loops			
				GFLOPS	AI		Vector I...	Efficiency	Gain Es...	
[loop in bench_stencil_ver4\$...	159.595s	15	V.	23.083	0.117		AVX2	100%	5.27x	4
[loop in bench_stencil_ver3\$...	159.953s	15	V.	16.274	0.117		AVX2	89%	3.55x	4
[loop in bench_stencil_ver2\$...	160.035s	16	V.	15.662	0.117		AVX2	80%	3.21x	4
[loop in bench_stencil_ver1\$...	159.307s	15	V.	10.218	0.117		AVX2	80%	3.21x	4
[loop in bench_stencil_ver0...	157.994s	1	S.	9.009	0.117					

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:29 bench_stencil_ver0\$omp\$parallel_for@25

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
25	#pragma omp parallel for	9.890s				
26	for(k=1;k<dim+1;k++){					
27	for(j=1;j<dim+1;j++){					
28	#pragma novector					
29	for(i=1;i<dim+1;i++){	102.403s		157.994s		
30	int ijk = i*iStride + j*jStride + k*kStride;					
31	new[ijk] = -6.0*old[ijk	53.651s				FMA
32	+ old[ijk-iStride]					
Selected (Total Time):		102.403s				

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

Cache-Aware Roofline

Intel Advisor 2017 interface showing performance analysis for a stencil computation. The window title is "NoMachine - NERSC" and the file path is "/global/cscratch1/sd/tk".

Elapsed time: 50.50s. Status: Vectorized, Not Vectorized. Smart Mode: OFF.

Summary | Survey & Roofline | Refinement Reports

Performance (GFLOPS) vs. Self Elapsed Time (s) graph. The y-axis is logarithmic (1 to 1000 GFLOPS) and the x-axis is logarithmic (0.01 to 10 s). The graph shows several dashed lines representing bandwidth limits: L1 Bandwidth: 5516.07 GB/sec, L2 Bandwidth: 1801.27 GB/sec, L3 Bandwidth: 502.16 GB/sec, and DRAM Bandwidth: 128.85 GB/sec. A data point is highlighted at approximately 0.1 s and 9.01 GFLOPS.

DP Vector FMA Peak: 843.06 GFLOPS
 DP Vector Add Peak: 210.96 GFLOPS
 Scalar Add Peak: 57.44 GFLOPS

bench_stencil_ver0\$omp\$parallel_for@25 stencil_v2.c:29
 Performance: 9.01 GFLOPS
 L1 Arithmetic Intensity: 0.12 FLOP/Byte
 Self Elapsed Time: 10.012 s
 Total Time: 157.994 s

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

File: stencil_v2.c:29 bench_stencil_ver0\$omp\$parallel_for@25

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
28	#pragma novector					
29	for(i=1; i<dim+1; i++){	102.403s		157.994s		
Selected (Total Time):		102.403s				

Self Elapsed Time: 10.012 s Total Time: 157.994 s

Advixe-gui | emacs-gtk@cori05-bond0.224 | cori :

Cache-Aware Roofline

Intel Advisor 2017 interface showing a Cache-Aware Roofline plot. The plot displays Performance (GFLOPS) on the y-axis (log scale, 1 to 1000) versus Self Elapsed Time (s) on the x-axis (log scale, 0.01 to 10). The plot includes several dashed lines representing bandwidth limits: L1 Bandwidth (5516.07 GB/sec), L2 Bandwidth (1801.27 GB/sec), L3 Bandwidth (502.16 GB/sec), and DRAM Bandwidth (128.85 GB/sec). A data point for the current configuration is highlighted with a yellow dot at approximately 0.12s and 10.22 GFLOPS. A tooltip for this point shows: `bench_stencil_ver1ompparallel_for@62 stencil_v2.c:65`, Performance: 10.22 GFLOPS, L1 Arithmetic Intensity: 0.12 FLOP/Byte, Self Elapsed Time: 10.206 s, and Total Time: 159.307 s. Other performance metrics shown include DP Vector FMA Peak (843.06 GFLOPS), DP Vector Add Peak (210.96 GFLOPS), and Scalar Add Peak (57.44 GFLOPS). The interface also shows a table of source code snippets and their execution times.

Source	Total Time	%	Loop/Function Time	%	Traits
64 for(j=1;j<dim+1;j++){	0.020s				
65 for(i=1;i<dim+1;i++){	0.684s		159.307s		
Selected (Total Time):			0.684s		

Cache-Aware Roofline

Intel Advisor 2017 interface showing performance analysis for a stencil computation. The window title is "NoMachine - NERSC" and the file path is "/global/cscratch1/sd/tk".

Elapsed time: 50.50s. Status: Vectorized, Not Vectorized. Smart Mode: OFF.

Summary | Survey & Roofline | Refinement Reports

Performance (GFLOPS) vs. Self Elapsed Time (s) graph. The y-axis ranges from 1 to 1000 GFLOPS, and the x-axis ranges from 0.01 to 10 s. The graph shows several performance curves and bandwidth limits:

- L1 Bandwidth: 5516.07 GB/sec?
- L2 Bandwidth: 1801.27 GB/sec?
- L3 Bandwidth: 502.16 GB/sec?
- DRAM Bandwidth: 128.85 GB/sec?
- DP Vector FMA Peak: 843.06 GFLOPS?
- DP Vector Add Peak: 210.96 GFLOPS?
- Scalar Add Peak: 57.44 GFLOPS?

Selected point: `bench_stencil_ver2ompparallel_for@102 stencil_v2.c:108`
 Performance: 15.66 GFLOPS
 L1 Arithmetic Intensity: 0.12 FLOP/Byte
 Self Elapsed Time: 10.438 s
 Total Time: 160.035 s

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

File: `stencil_v2.c:108 bench_stencil_ver2ompparallel_for@102`

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
107	<code>for(j=jLo; j<jLo+16; j++){</code>	0.092s				
108	<code>for(i=0; i<dim; i++){</code>	1.500s		160.035s		
Selected (Total Time):		1.500s				

Cache-Aware Roofline

Intel Advisor 2017 interface showing a Cache-Aware Roofline plot. The plot displays performance (GFLOPS) on a logarithmic scale versus self-elapsed time (s) on a logarithmic scale. The plot includes several dashed lines representing bandwidth limits: L1 Bandwidth (5516.07 GB/sec), L2 Bandwidth (1801.27 GB/sec), L3 Bandwidth (502.16 GB/sec), and DRAM Bandwidth (128.85 GB/sec). The current performance is 16.27 GFLOPS, which is significantly below the DRAM bandwidth limit. A tooltip for the selected point shows: bench_stencil_ver3\$omp\$parallel_for@146 stencil_v2.c:152, Performance: 16.27 GFLOPS, L1 Arithmetic Intensity: 0.12 FLOP/Byte, Self Elapsed Time: 10.144 s, Total Time: 159.953 s.

Summary: Self Elapsed Time: 10.144 s, Total Time: 159.953 s

Source	Top Down	Code Analytics	Assembly	Recommendations	Why No Vectorization?			
File: stencil_v2.c:152 bench_stencil_ver3\$omp\$parallel_for@146								
Lin.	Source			Total Time	%	Loop/Function Time	%	Traits
151	for(j=jLo; j<jLo+16; j++){			0.016s				
152	for(i=0; i<jStride; i++){			0.708s		159.953s		
Selected (Total Time):				0.708s				

Cache-Aware Roofline

Intel Advisor 2017 interface showing performance analysis for a stencil computation. The window title is "NoMachine - NERSC" and the file path is "/global/cscratch1/sd/tk".

Elapsed time: 50.50s. Status: Vectorized, Not Vectorized. Smart Mode: OFF.

Summary | Survey & Roofline | Refinement Reports

Performance (GFLOPS) vs. Self Elapsed Time (s) graph. Y-axis: 1, 10, 100, 1000 GFLOPS. X-axis: 0.01, 0.1, 1, 10 s. Legend: L1 Bandwidth: 5516.07 GB/sec?, L2 Bandwidth: 1801.27 GB/sec?, L3 Bandwidth: 502.16 GB/sec?, DRAM Bandwidth: 128.85 GB/sec?. Peak values: DP Vector FMA Peak: 843.06 GFLOPS?, DP Vector Add Peak: 210.96 GFLOPS?, Scalar Add Peak: 57.44 GFLOPS?.

Selected benchmark: `bench_stencil_ver4ompparallel_for@193 stencil_v2.c:201`. Performance: 23.08 GFLOPS. Self Elapsed Time: 10.004 s. Total Time: 159.595 s.

Source: Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

File: `stencil_v2.c:201 bench_stencil_ver4ompparallel_for@193`

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
200	<code>#pragma vector nontemporal</code>					
201	<code>for(i=0; i<jStride; i++){</code>	3.636s		159.595s		
Selected (Total Time):		3.636s				

DRAM Roofline*

Intel Advisor 2018 interface showing a DRAM Roofline analysis. The graph plots Performance (GFLOPS) on the y-axis (0.66 to 1688.18) against Self Elapsed Time (s) on the x-axis (0.05 to 5). The roofline is bounded by L1, L2, L3, and DRAM bandwidths. A tooltip for a specific loop provides performance and intensity metrics.

Performance (GFLOPS)

1688.18

0.66

0.05

5

Self Elapsed Time: 0.000 s Total Elapsed Time: 10.000 s

Intensity (FLOP/Byte)

Bandwidths:

- L1 Bandwidth: 1.1e+4 GB/sec
- L2 Bandwidth: 3542.01 GB/sec
- L3 Bandwidth: 1003.46 GB/sec
- DRAM Bandwidth: 128.58 GB/sec

Peaks:

- DP Vector FMA Peak: 1688.18 GFLOPS
- DP Vector Add Peak: 422.07 GFLOPS
- Scalar Add Peak: 115.16 GFLOPS

Loop Performance:

[loop in bench_stencil_ver0\$omp\$parallel_for@25 at stencil_v2.c:26]

- Total Performance: 5.45 GFLOPS
- Total L1 Arithmetic Intensity: 0.17 FLOP/Byte
- Self Elapsed Time: 0.000 s
- Total Elapsed Time: 10.000 s

Source Code:

```
File: stencil_v2.c:26 bench_stencil_ver0$omp$parallel_for@25
```

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
24	while(ElapsedTime < TIME){					
25	#pragma omp parallel for					
26	for(k=1;k<dim+1;k++){			159417.000ms		
27	for(j=1;j<dim+1;j++){					
Selected (Total Time):		0ms				

DRAM Roofline*

Intel Advisor 2018

Elapsed time: 50.40s

Vectorized Not Vectorized MKL

Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Performance (GFLOPS)

1688.18

0.66

L1 Bandwidth: 1.1e+4 GB/sec

L2 Bandwidth: 3542.01 GB/sec

L3 Bandwidth: 1003.46 GB/sec

DRAM Bandwidth: 128.58 GB/sec

DP Vector FMA Peak: 1688.18 GFLOPS

DP Vector Add Peak: 422.07 GFLOPS

Scalar Add Peak: 115.16 GFLOPS

Self Elapsed Time: 0.000 s Total Elapsed Time: 9.972 s

Intensity (FLOP/Byte)

Use Single-Threaded Roofs Show Hierarchical Data

[loop in bench_stencil_ver1\$omp\$parallel_for@62 at stencil_v2.c:63]

Total Performance: 10.18 GFLOPS

Total L1 Arithmetic Intensity: 0.17 FLOP/Byte

Self Elapsed Time: 0.000 s

Total Elapsed Time: 9.972 s

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:63 bench_stencil_ver1\$omp\$parallel_for@62

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
61	while(ElapsedTime < TIME){					
62	#pragma omp parallel for					
63	for(k=1;k<dim+1;k++){			159453.000ms		
64	for(j=1;j<dim+1;j++){					
Selected (Total Time):		0ms				

DRAM Roofline*

Intel Advisor 2018

Elapsed time: 50.40s

Vectorized Not Vectorized MKL

Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Performance (GFLOPS)

1688.18

0.66

L1 Bandwidth: 1.1e+4 GB/sec

L2 Bandwidth: 3542.01 GB/sec

L3 Bandwidth: 1003.46 GB/sec

DRAM Bandwidth: 128.58 GB/sec

DP Vector FMA Peak: 1688.18 GFLOPS

DP Vector Add Peak: 422.07 GFLOPS

Scalar Add Peak: 115.16 GFLOPS

[loop in bench_stencil_ver2\$omp\$parallel_for@102 at stencil_v2.c:103]

Total Performance: 15.28 GFLOPS

Total L1 Arithmetic Intensity: 0.28 FLOP/Byte

Self Elapsed Time: 0.008 s

Total Elapsed Time: 9.964 s

Self Elapsed Time: 0.008 s Total Elapsed Time: 9.964 s

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:103 bench_stencil_ver2\$omp\$parallel_for@102

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
101	while(ElapsedTime < TIME){					
102	#pragma omp parallel for schedule(static,1)	16.002ms				
103	for(tile=0;tile<jTiles*kTiles;tile++){	159651.000ms				
104	int kLo = 16*(tile/jTiles);					Divisi...
Selected (Total Time):		0ms				

DRAM Roofline*

Intel Advisor 2018

Elapsed time: 50.40s

Vectorized Not Vectorized MKL

Smart Mode

FILTER: All Modules All Sources Loops And Functions All Threads

Summary Survey & Roofline Refinement Reports

Performance (GFLOPS)

1688.18

0.66

0.05

Self Elapsed Time: 0.000 s Total Elapsed Time: 9.926 s

L1 Bandwidth: 1.1e+4 GB/sec

L2 Bandwidth: 3542.01 GB/sec

L3 Bandwidth: 1003.46 GB/sec

DRAM Bandwidth: 128.58 GB/sec

DP Vector FMA Peak: 1688.18 GFLOPS

DP Vector Add Peak: 422.07 GFLOPS

Scalar Add Peak: 115.16 GFLOPS

[loop in bench_stencil_ver3\$omp\$parallel_for@146 at stencil_v2.c:146]
Total Performance: 15.67 GFLOPS
Total L1 Arithmetic Intensity: 0.28 FLOP/Byte
Self Elapsed Time: 0.000 s
Total Elapsed Time: 9.926 s

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: stencil_v2.c:146 bench_stencil_ver3\$omp\$parallel_for@146

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
144	StartTime = omp_get_wtime();					
145	while(ElapsedTime < TIME){					
146	#pragma omp parallel for schedule(static,1)	43.998ms		159807.000ms		
147	for(tile=0;tile<jTiles*kTiles;tile++){					
Selected (Total Time):		43.998ms				

Advixe-gui emacs-gtk@cori05-bond0.224 cori :

DRAM Roofline*

Intel Advisor 2018 interface showing a DRAM Roofline analysis. The window title is "NoMachine - NERSC" and the file path is "/global/cscratch1/sd/tk".

Elapsed time: 50.40s. Vectorized, Not Vectorized, MKL. Smart Mode OFF.

FILTER: All Modules, All Sources, Loops And Functions, All Threads.

Summary, Survey & Roofline, Refinement Reports.

Performance (GFLOPS) graph showing L1 Bandwidth (1.1e+4 GB/sec), L2 Bandwidth (3542.01 GB/sec), L3 Bandwidth (1003.46 GB/sec), and DRAM Bandwidth (128.58 GB/sec). Performance peaks are shown for DP Vector FMA (1688.18 GFLOPS), DP Vector Add (422.07 GFLOPS), and Scalar Add (115.16 GFLOPS).

Self Elapsed Time: 0.000 s, Total Elapsed Time: 9.956 s.

[loop in bench_stencil_ver4\$omp\$parallel_for@193 at stencil_v2.c:193]
Total Performance: 18.98 GFLOPS
Total L1 Arithmetic Intensity: 0.41 FLOP/Byte
Self Elapsed Time: 0.000 s
Total Elapsed Time: 9.956 s

Source: Top Down, Code Analytics, Assembly, Recommendations, Why No Vectorization?

File: stencil_v2.c:193 bench_stencil_ver4\$omp\$parallel_for@193

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
191	StartTime = omp_get_wtime();					
192	while(ElapsedTime < TIME){					
193	#pragma omp parallel for schedule(static,1)	8.004ms		160161.000ms		
194	for(tile=0;tile<jTiles*kTiles;tile++){					
Selected (Total Time):		8.004ms				

Little's Law

Applied to Memory

- Consider a CPU with 100GB/s of bandwidth and 100ns memory latency.
- Little's law states that we must **express 10KB of concurrency** (independent memory operations) to the memory subsystem to attain peak performance
- Solution #1: use multiple cores or threads to satisfy the requisite MLP.
- Solution #2: express the memory access pattern in a streaming fashion in order to engage the prefetchers.

Applied to FPUs

- consider a CPU with 2 FPU's each with a 4-cycle latency.
- Little's law states that we must express 8-way ILP to fully utilize the machine.
- Solution #1: rely on OOO to find parallelism across loop iterations.
- Solution #2: unroll/jam the code to express 8 **independent** FP operations.
- Note, simply unrolling dependent operations (e.g. reduction) does not increase ILP. It simply amortizes loop overhead.