Introduction to the Roofline Model

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

Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This 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.

Performance Models and Tools

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

Performance Models / Simulators

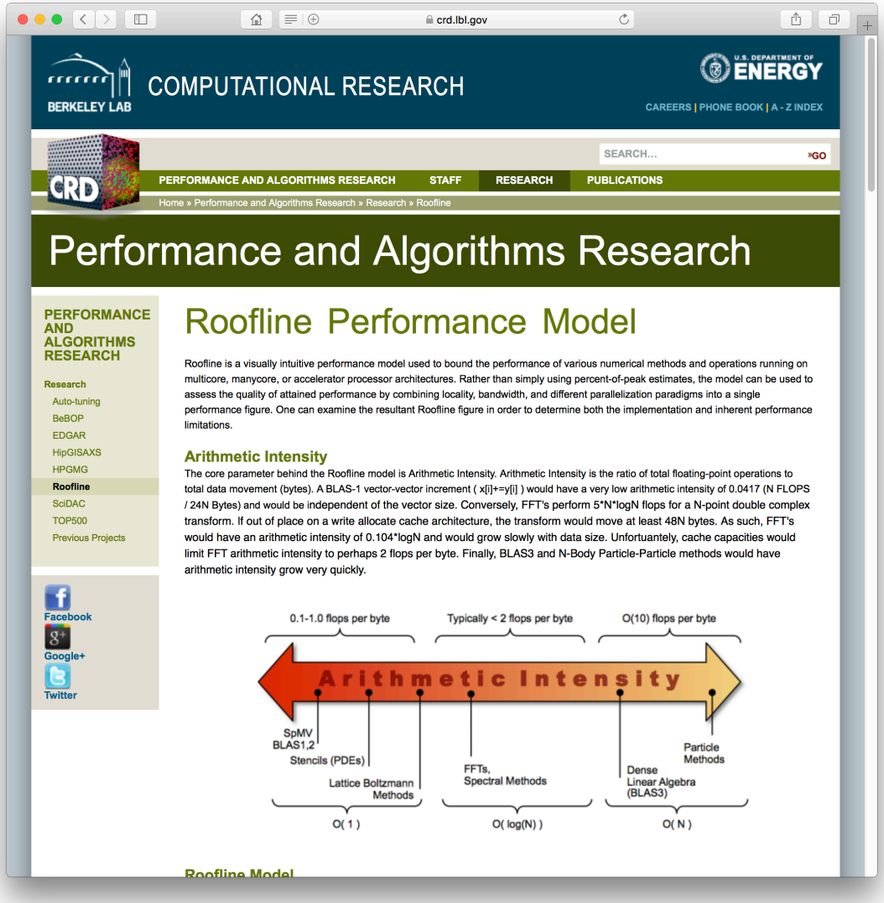
- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)

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

- Effectively latency hiding has resulted in a shift from a latency-limited computing regime to a **throughput-limited computing regime**

Roofline Model

- The **Roofline Model** is a throughput-oriented performance model…
  - Tracks rates not time
  - Augmented with Little's Law
    (concurrency = latency*bandwidth)
  - Independent of ISA and architecture
    (applies to CPUs, GPUs, Google TPUs[1], etc…)
- Three Components:
  - Machine Characterization
    (realistic performance potential of the system)
  - Application Execution Monitoring
  - Theoretical Application Bounds
    (how well could my app perform with perfect compilers, caches, overlap, …)

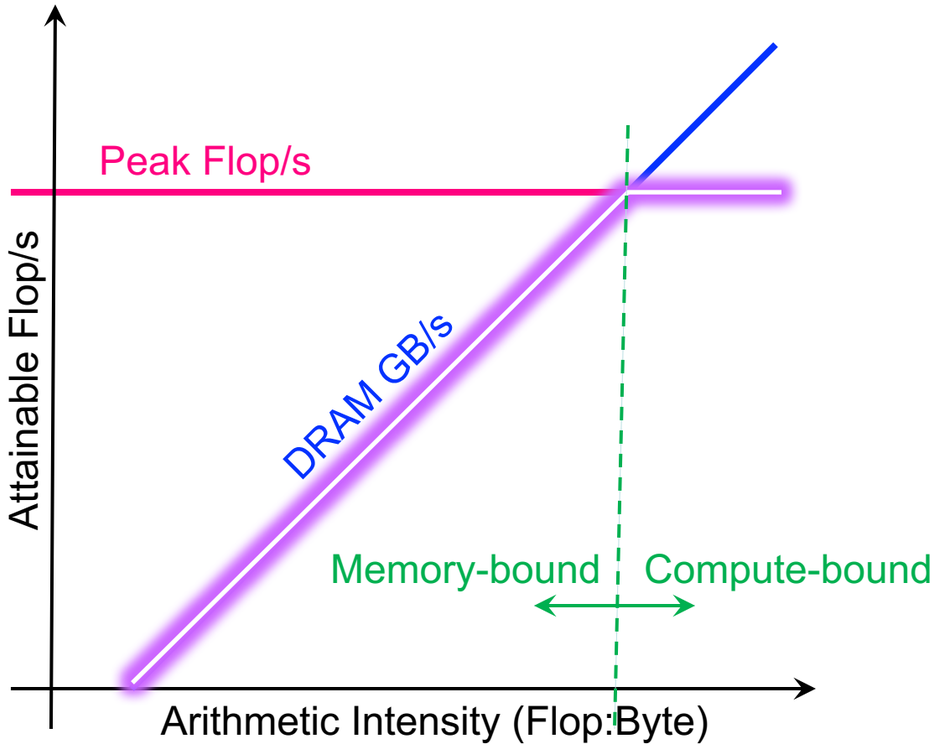


https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

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

(DRAM) Roofline

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



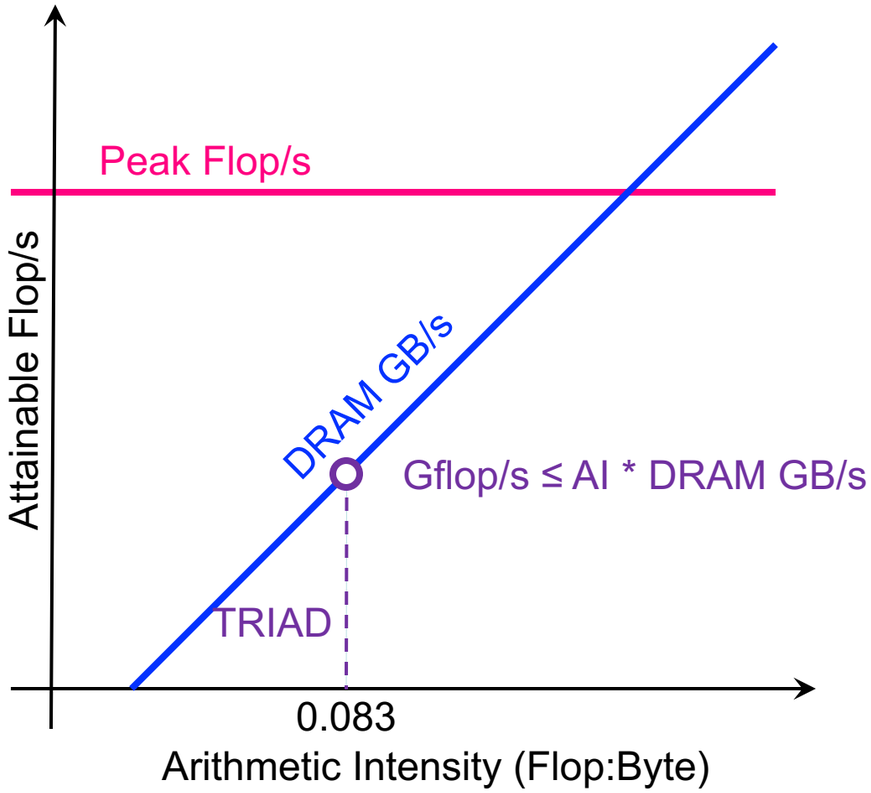
Roofline Example #1

- 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**

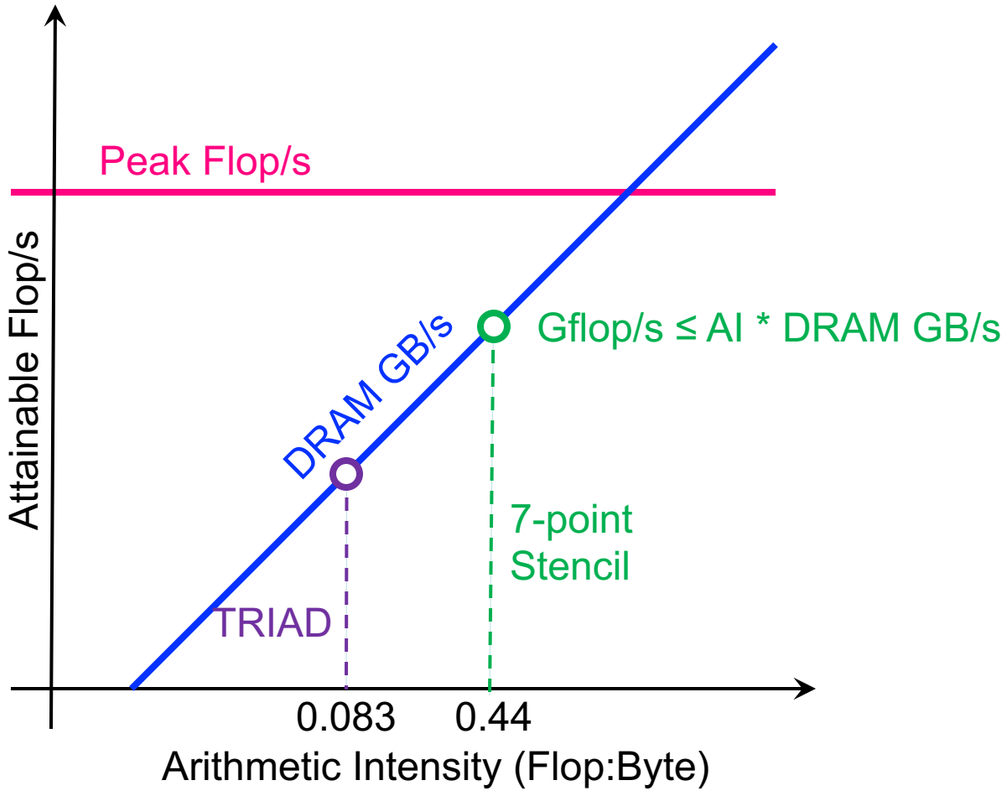


Roofline Example #2

- 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,**

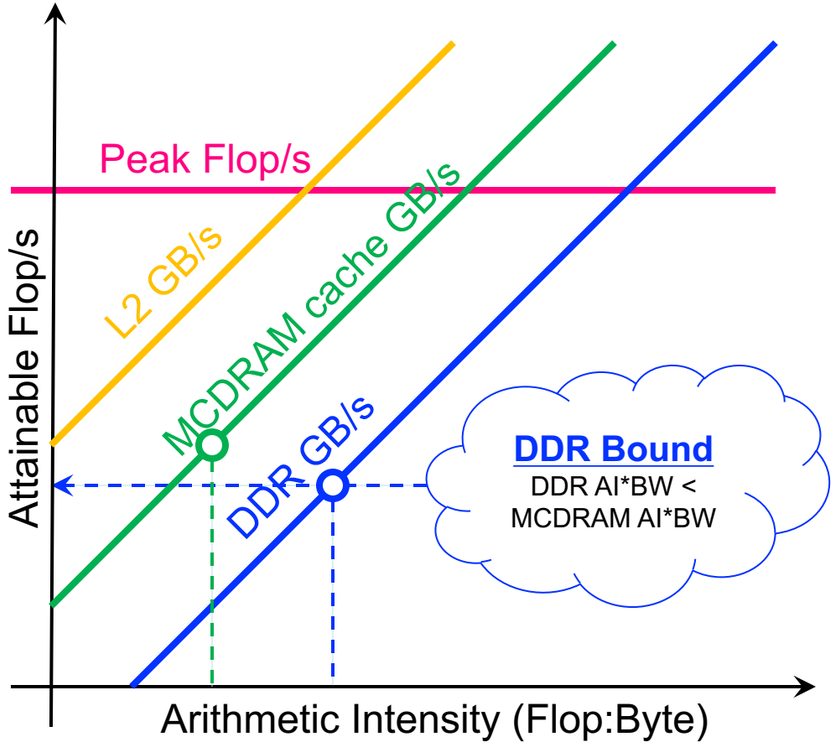
    but 5x the flop rate

```
#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];
}}}
```



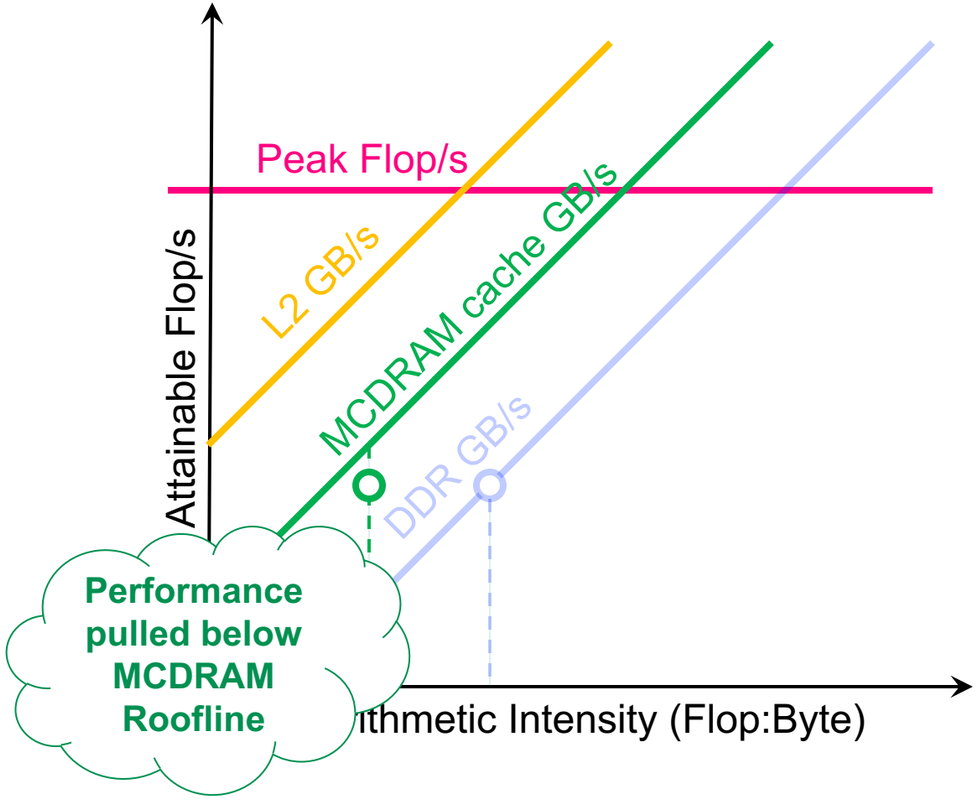
Hierarchical Roofline

- 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)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's and multiple bounds
  - A kernel could be DDR-limited…



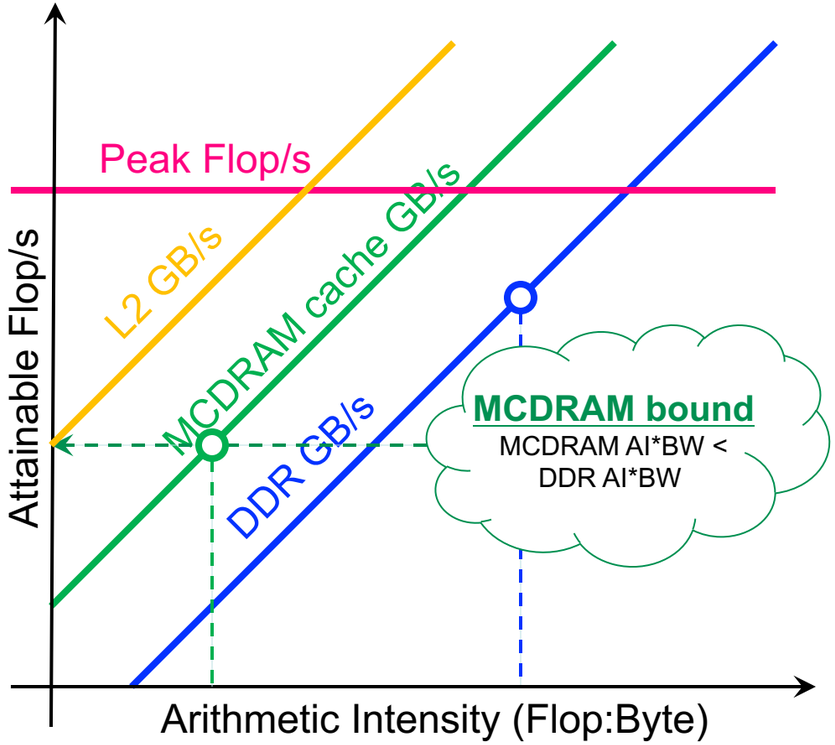
Hierarchical Roofline

- 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)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's and multiple bounds
  - A kernel could be DDR-limited…



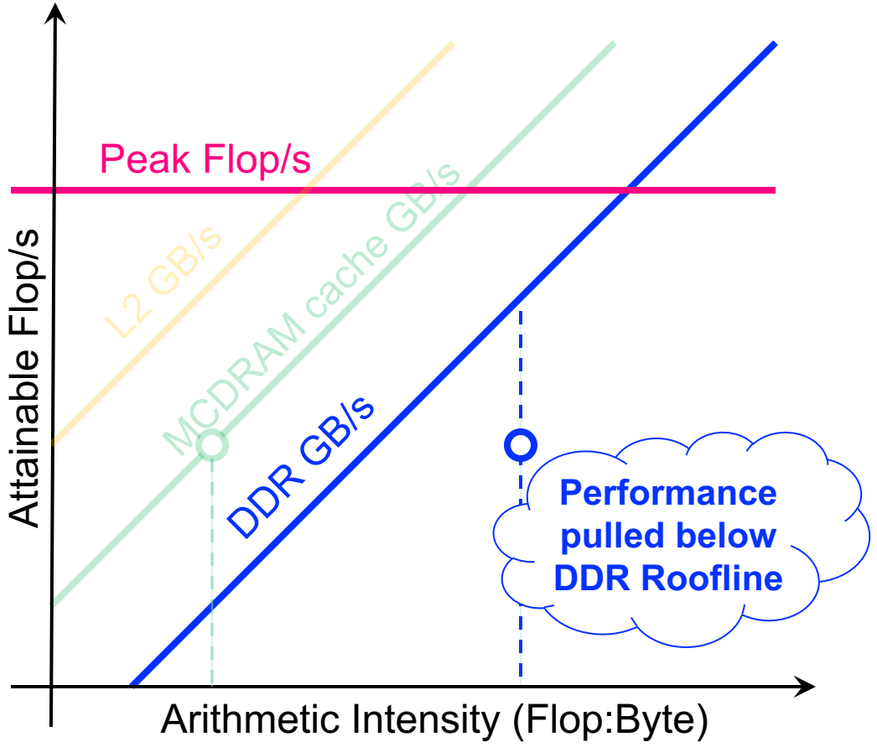
Hierarchical Roofline

- 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)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's and multiple bounds
  - A kernel could be DDR-limited…
  - **or MCDRAM-limited depending on relative bandwidths and AI's**



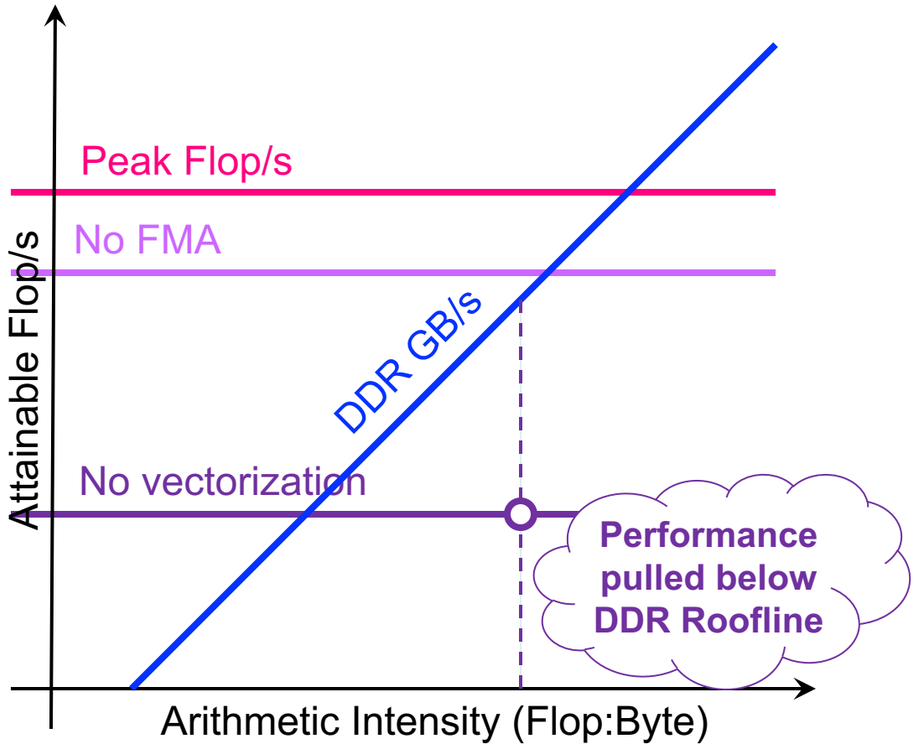
Hierarchical Roofline

- 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)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's and multiple bounds
  - A kernel could be DDR-limited…
  - **or MCDRAM-limited depending on relative bandwidths and AI's**



Data, Instruction, Thread-Level Parallelism…

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



Prior Roofline Efforts

Basic Roofline Modeling

Machine Characterization

Potential of my target system

- How does my system respond to a lack of FMA, DLP, ILP, TLP?
- How does my system respond to reduced AI (i.e. memory/cache bandwidth)?
- How does my system respond to NUMA, strided, or random memory access patterns?
- …

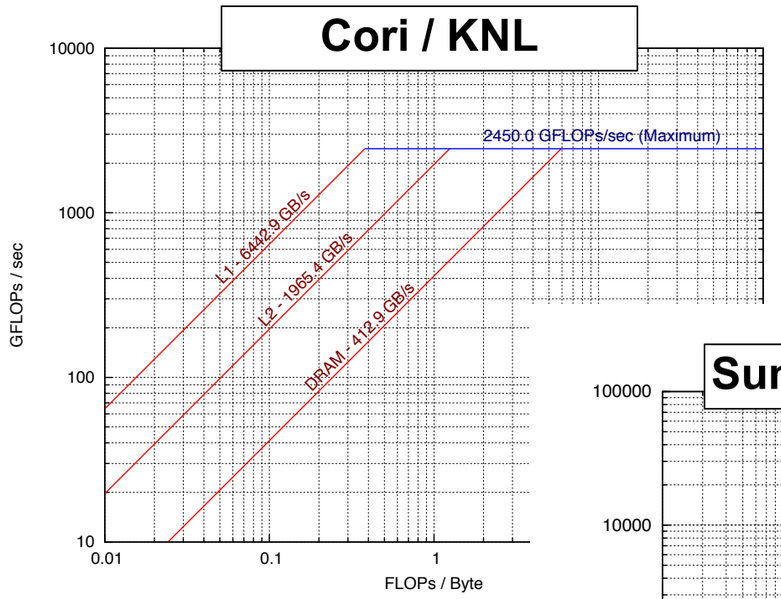
Application Instrumentation

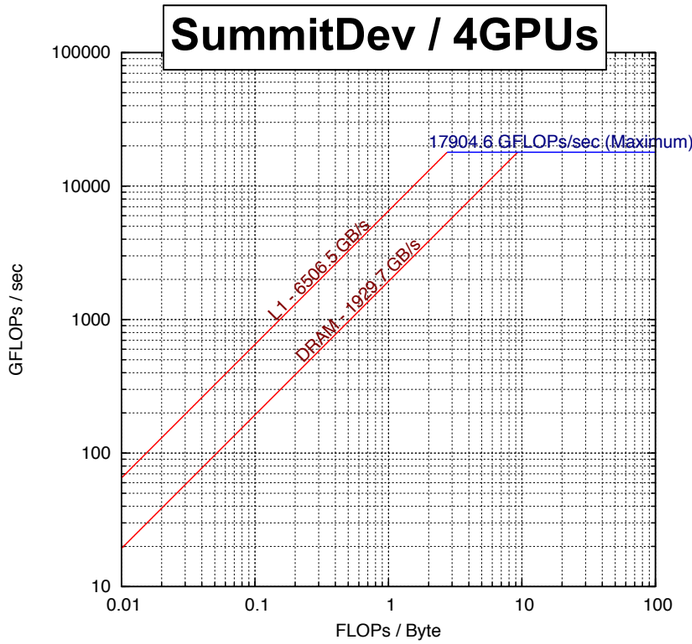
Properties of my app's execution

- What is my app/kernel's actual AI?
- How does AI vary with memory level ?
- How well does my app vectorize?
- Does my app use FMA?
- ...

Machine Characterization for Roofline

- How fast is my system?
- Challenges:
  - Too many systems; new ones each year
  - Voluminous documentation on each
  - Real performance often less than **"Marketing Numbers"**
  - **Compilers can "give up" on big loops**
- Empirical Roofline Toolkit (ERT)
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory
  - **MPI+OpenMP/CUDA == multiple GPUs**





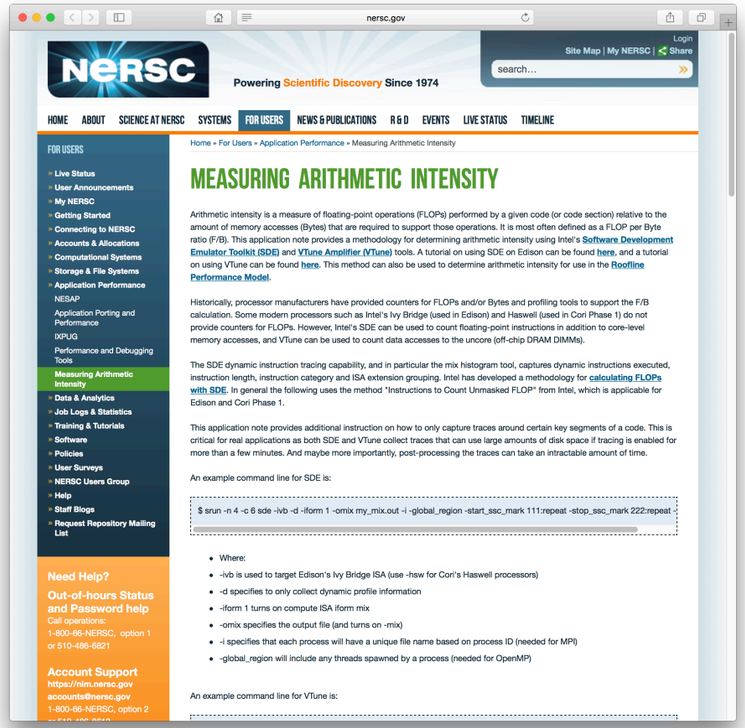
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Application Instrumentation Can Be Hard…

- ✘ Flop Counters can be broken/missing in production processors
- ✘ Vectorization/Masking can complicate counting Flop's
- ✘ Counting Loads and Stores is a poor proxy for DRAM data movement as they don't capture cache reuse
- ✘ Counting L1 misses is a poor proxy for data movement as they don't account for speculative HW stream prefetching.
- ✘ DRAM counters (Uncore PMU) might be accurate, but are privileged and thus nominally inaccessible in user mode
- ✘ OS/kernel changes must be approved by vendor (e.g. Cray) and the center (e.g. NERSC)

Initial NERSC Application Roofline Efforts

- Goal: Characterize applications running on NERSC production systems (Cori/KNL,HSW)
- Limited by available tools/permissions on Cori…
  - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
  - Used **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
- Produced 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…



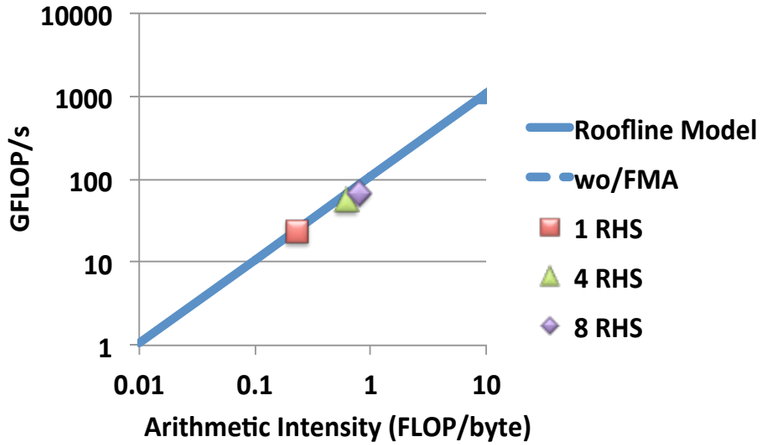
http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

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)

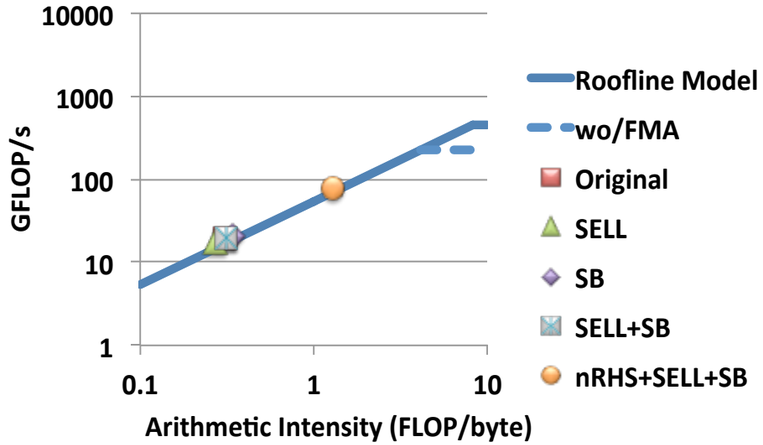
BERKELEY LAB

Initial Roofline Analysis of NESAP Codes

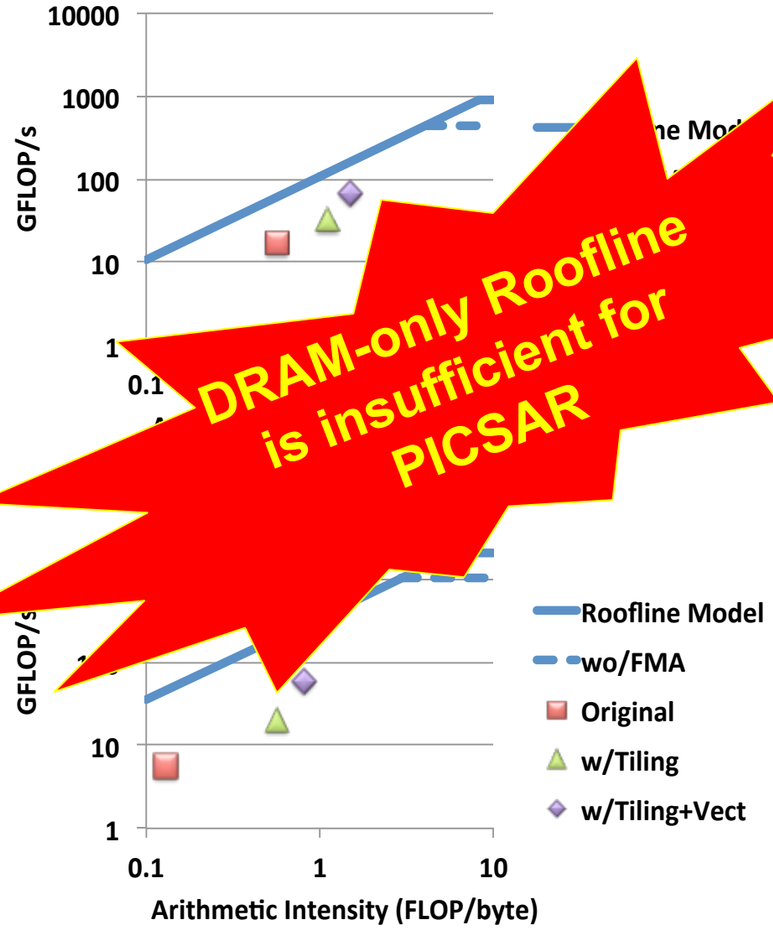
MFDn



EMGeo

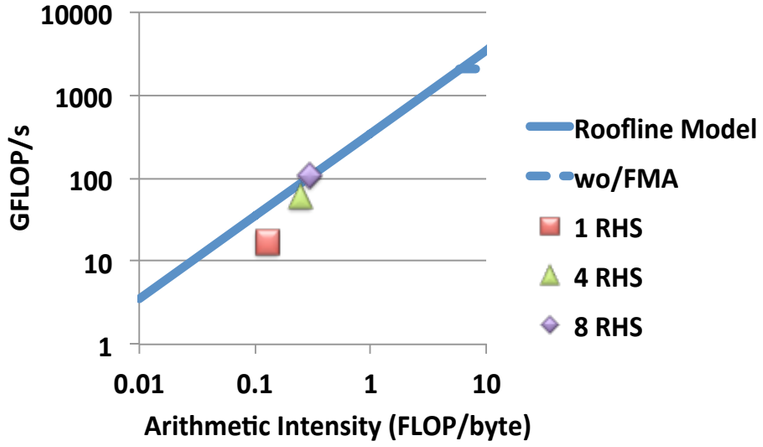


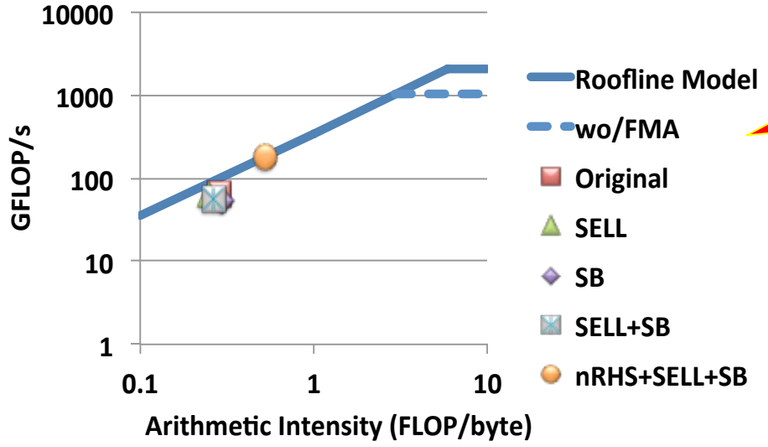
PICSAR



2P HSW

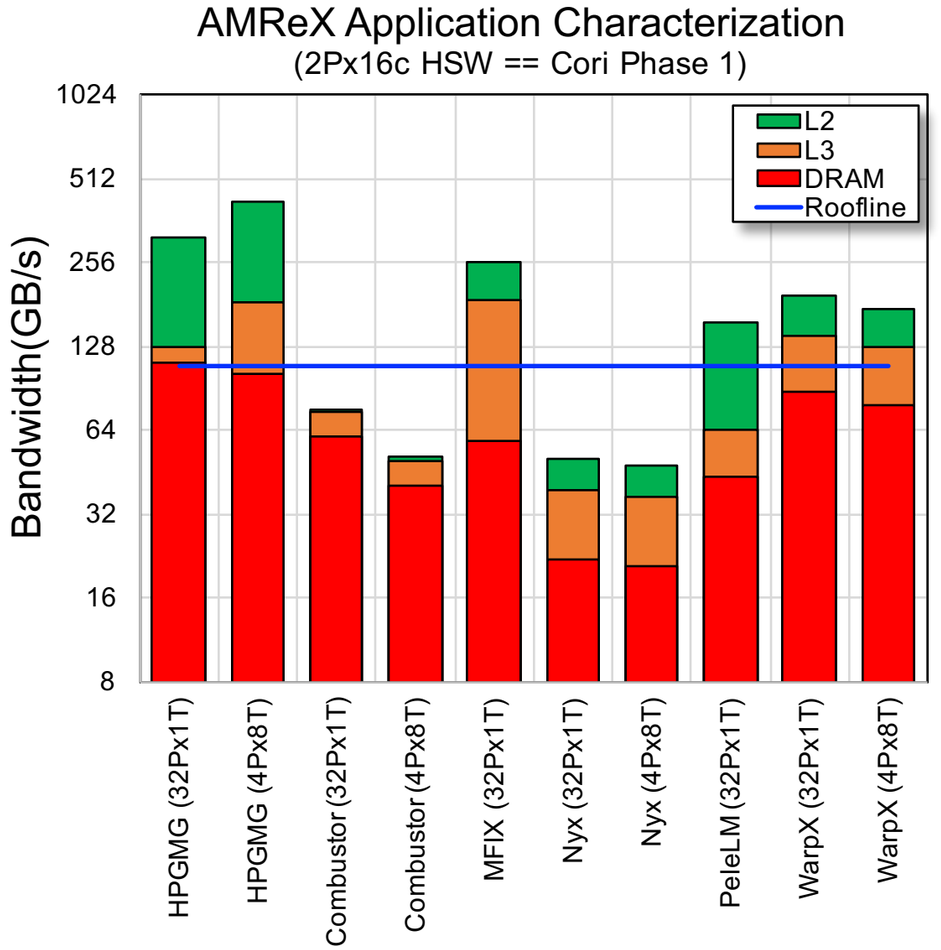
KNL





Additional Experimentation with LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters…
  - ✓ **Works on NERSC production systems**
  - ✓ Minimal overhead (<1%)
  - ✓ Scalable in distributed memory
  - ✓ Fast, high-level characterization at scale
  - ✓ Regions of interest can be manually marked and profiled
  - ✘ **No detailed timing breakdown or optimization advice**
  - ✘ **Limited by quality of underlying performance counters (garbage in/garbage out)**



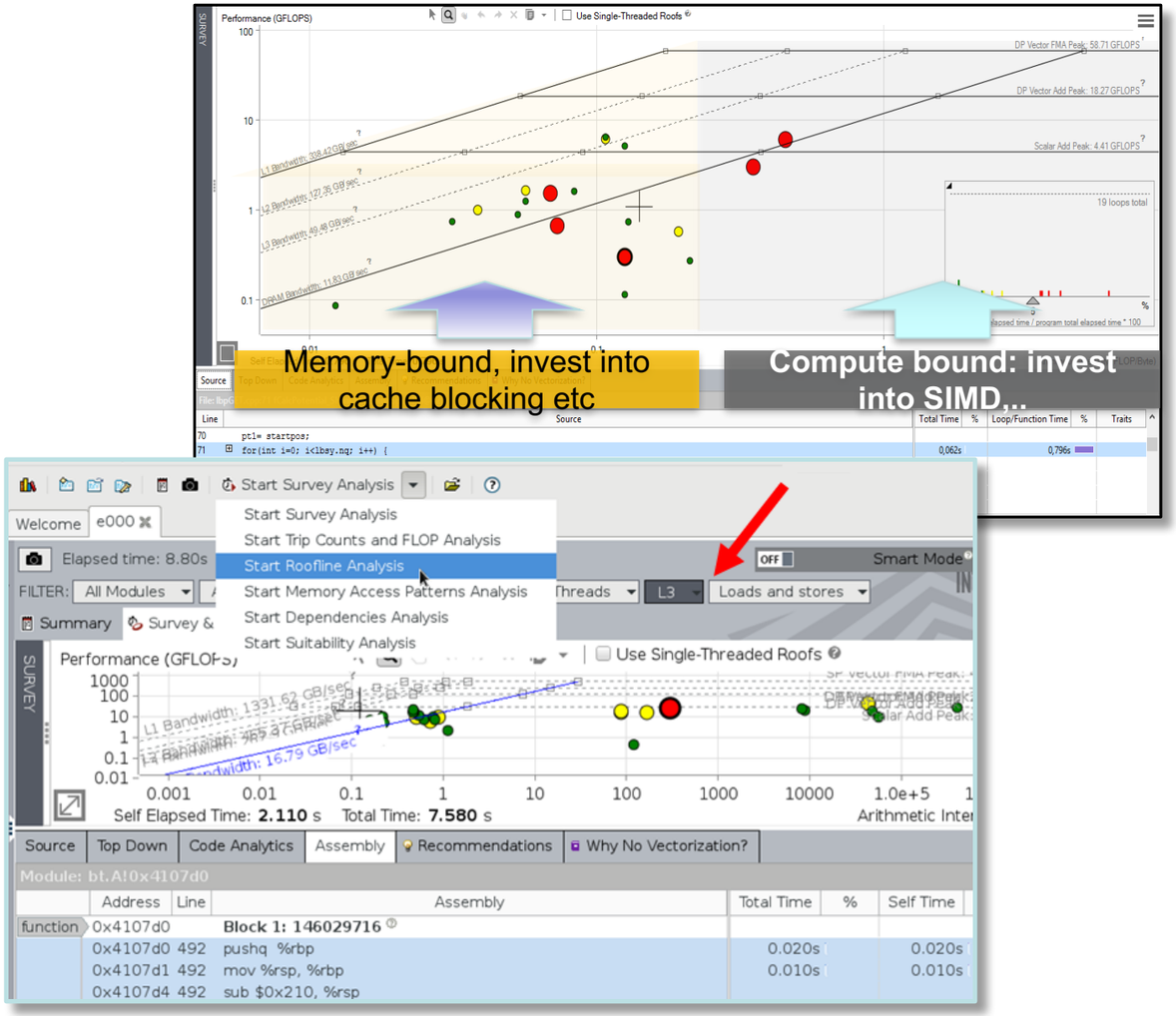
https://github.com/RRZE-HPC/likwid

Need a integrated solution…

- Having to compose VTune, SDE, and graphing tools to generate Roofline models worked correctly and benefitted NESAP, but …
  - ✘ forced users to learn/run multiple tools
  - ✘ forced users to instrument routines of interest in their application
  - ✘ forced users to manually parse/compose/graph the output
  - ✘ lacked integration with compiler/debugger/disassembly
- LIKWID was much easier to use, faster, and more scalable, but …
  - ✘ forces users to manually instrument routines of interest
  - ✘ forces users to manually parse/compose/graph the output
  - ✘ lacks integration with compiler/debugger/disassembly
- **CRD/NERSC wanted a more integrated solution for Roofline…**

Intel Advisor

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



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

Background:

Hierarchical Roofline vs. Cache-Aware Roofline

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
- 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 DRAM-only Roofline (PMU) has also been integrated into 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.
This version will be made available during the hands-on component of this tutorial.

Hierarchical Roofline

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

Cache-Aware Roofline

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

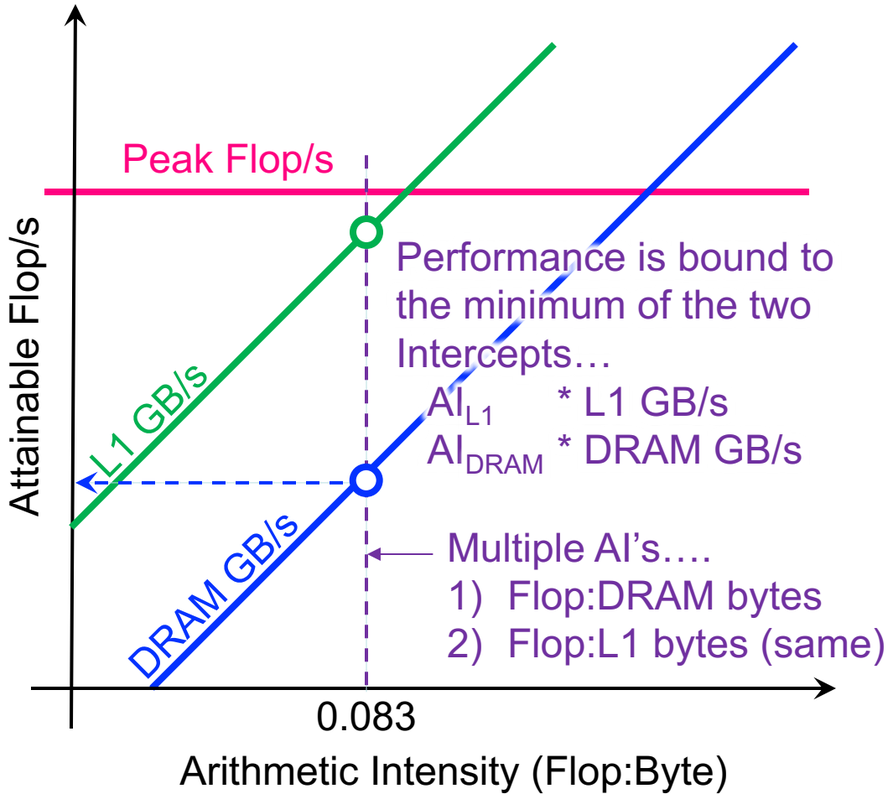
Example: STREAM

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

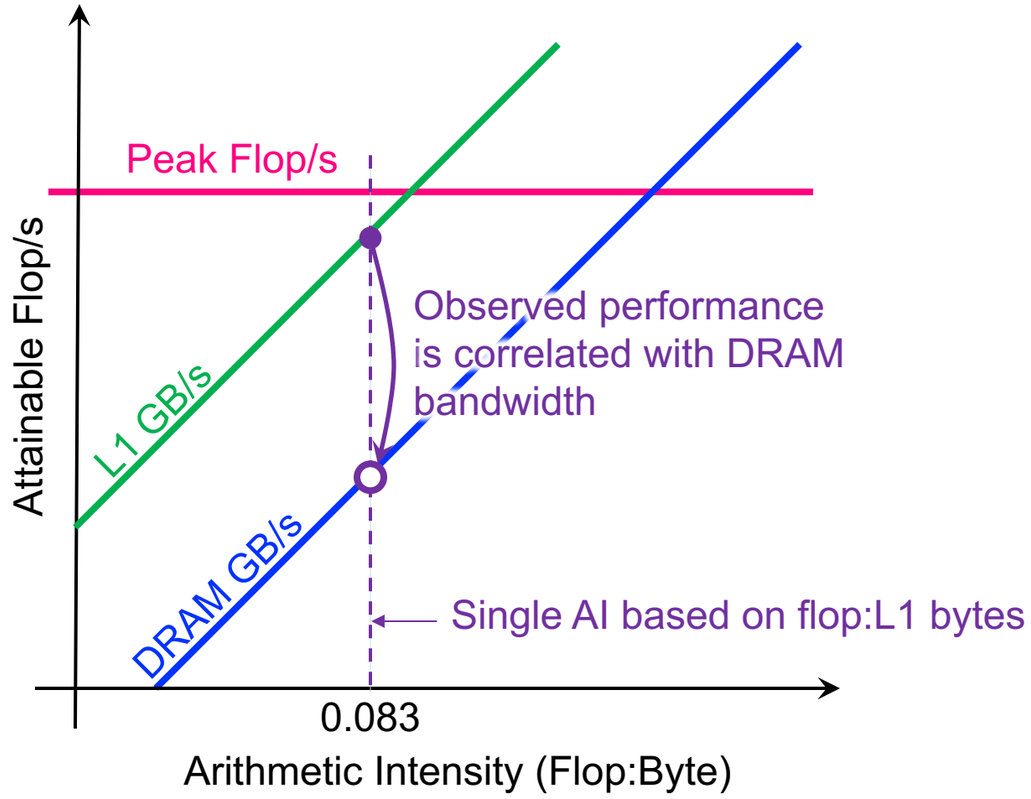
```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



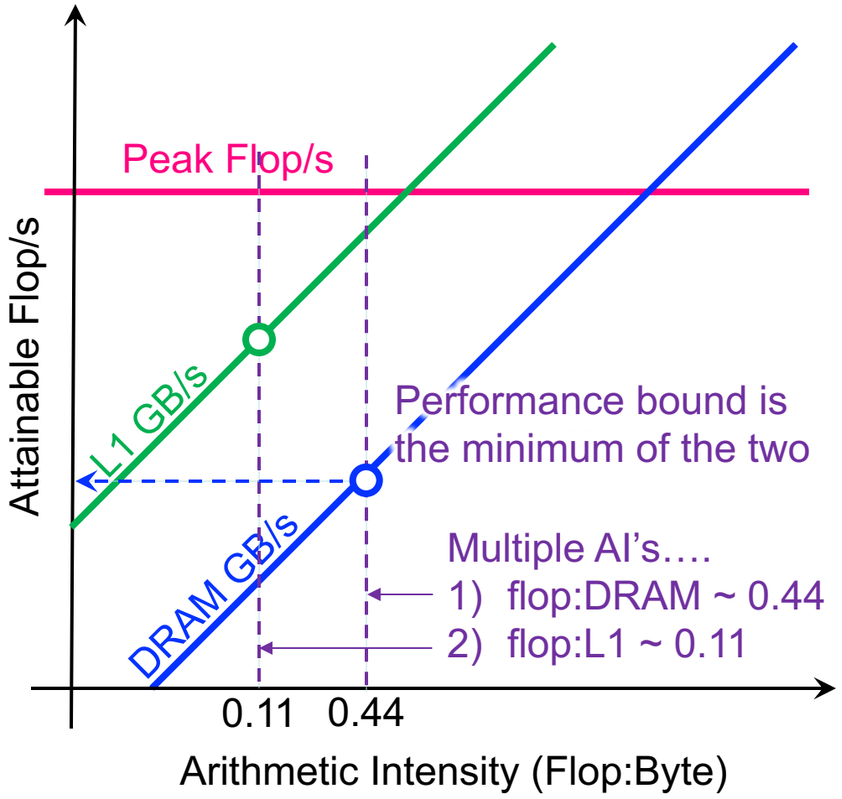
Example: 7-point Stencil (Small Problem)

- 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

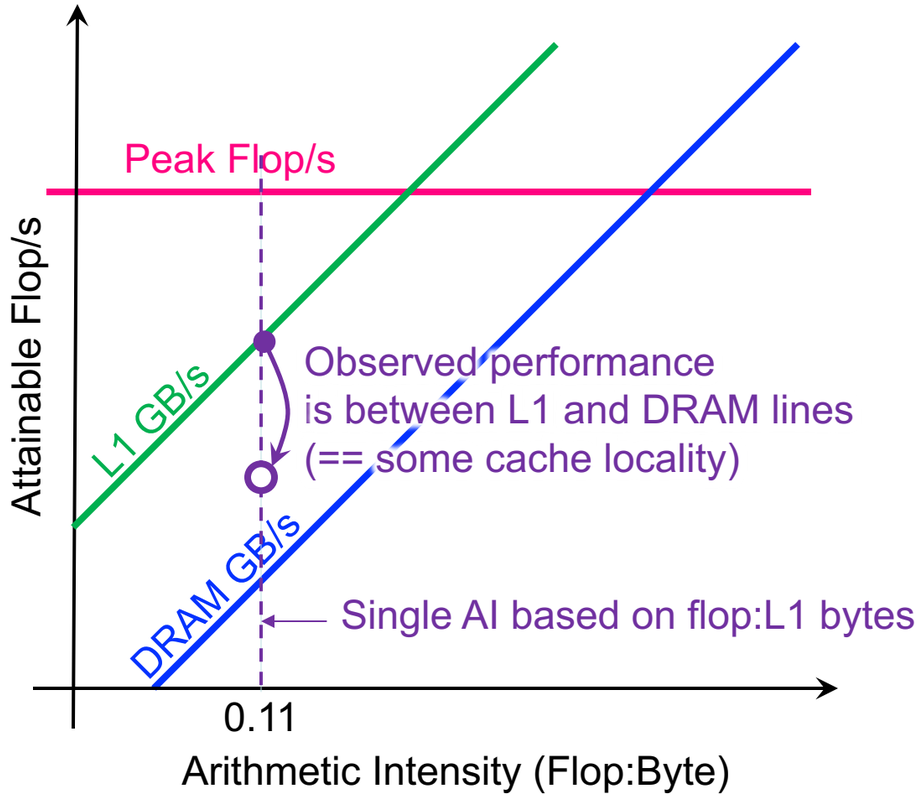
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk          ]
                 + old[ijk-1       ]
                 + old[ijk+1       ]
                 + old[ijk-jStride]
                 + old[ijk+jStride]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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

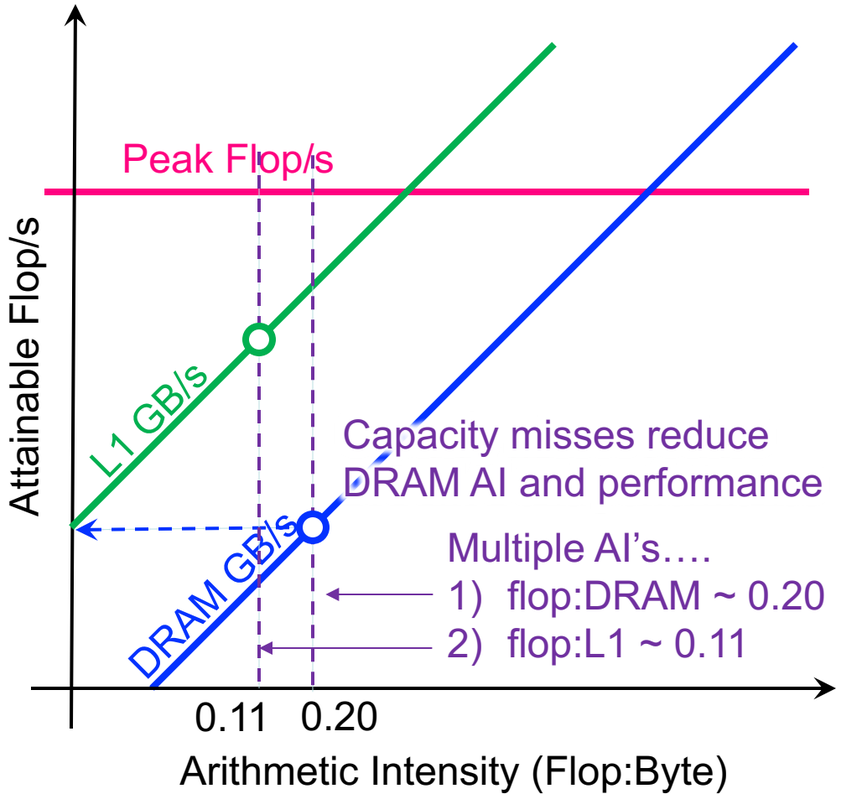


Cache-Aware Roofline

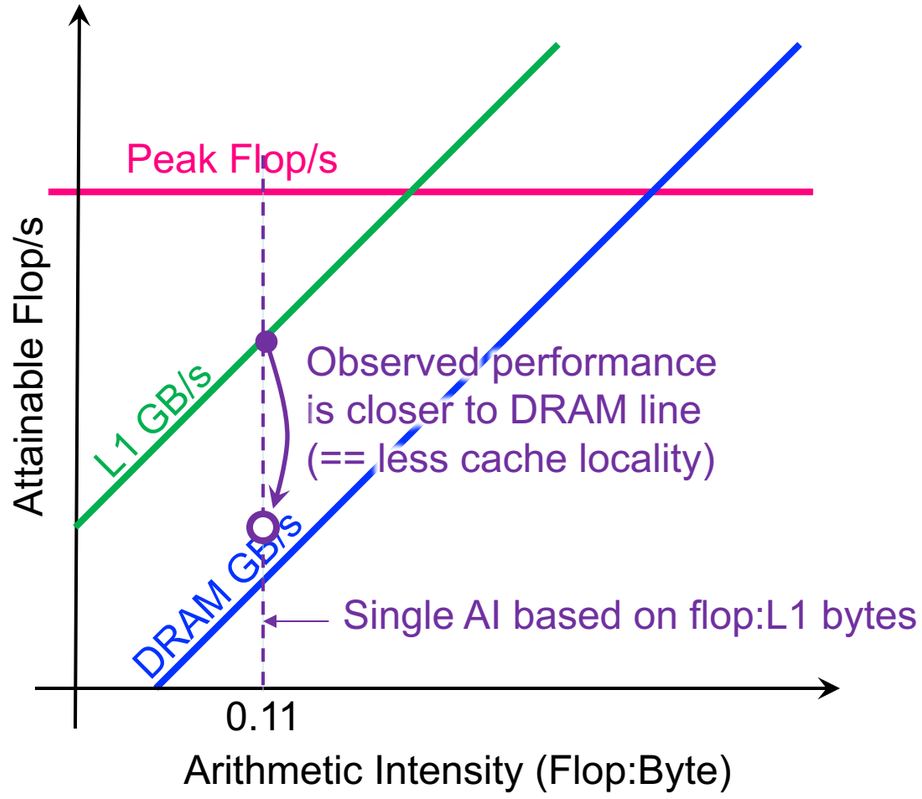


Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

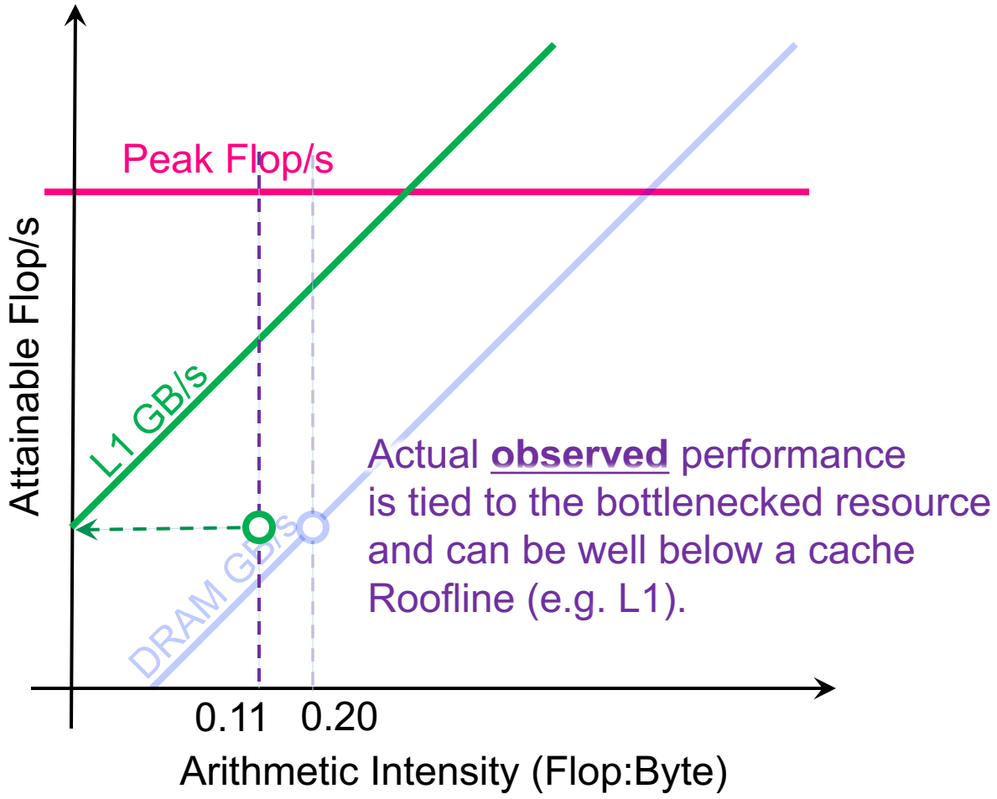


Cache-Aware Roofline

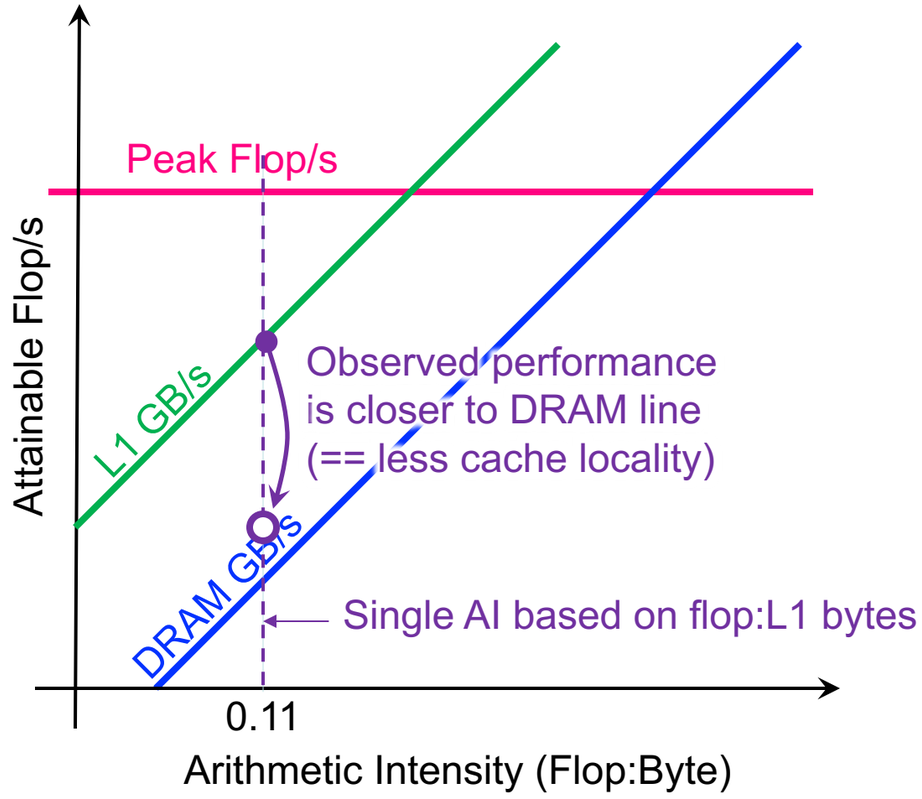


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

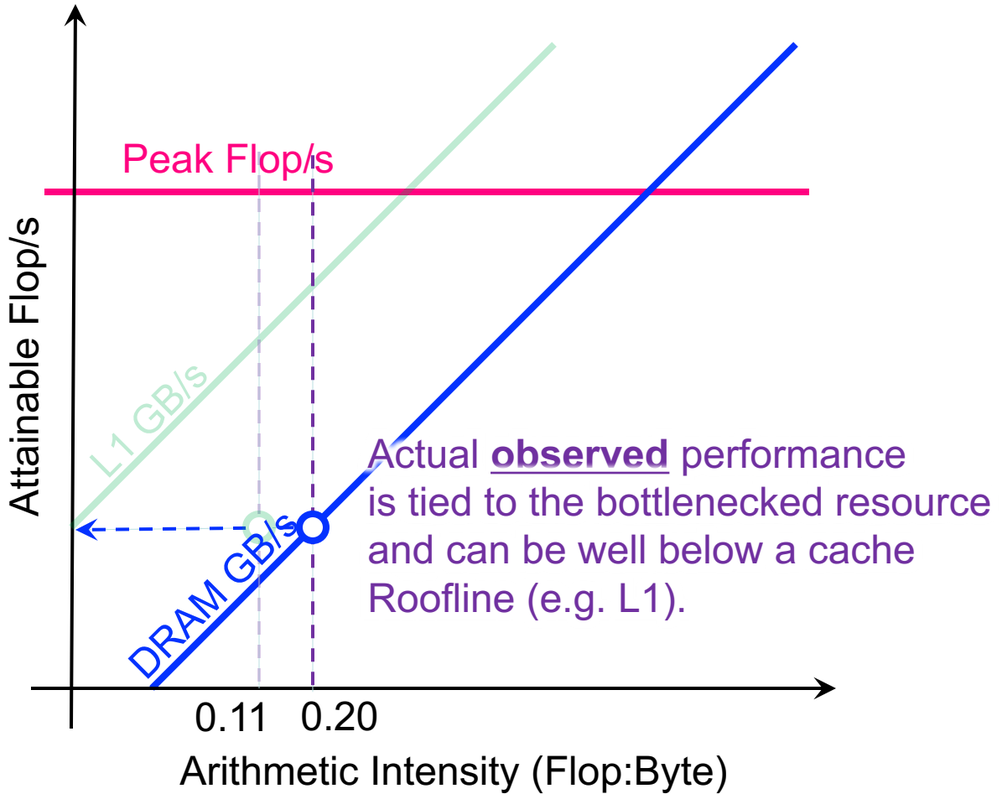


Cache-Aware Roofline

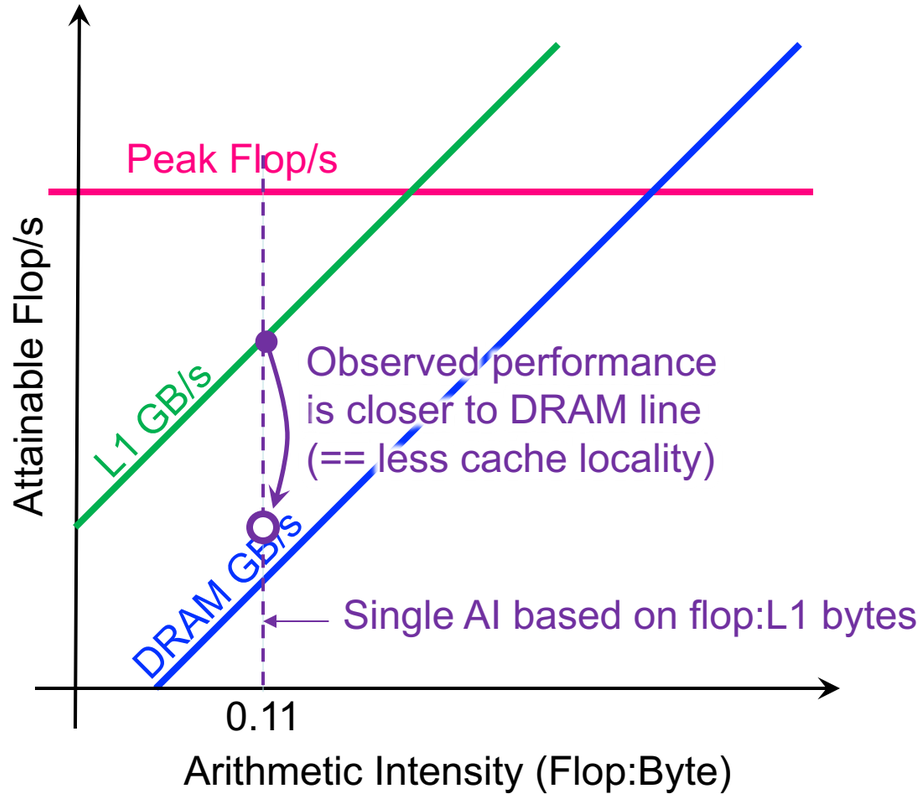


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Cache-Aware Roofline



Questions?