Parallelism and Performance

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

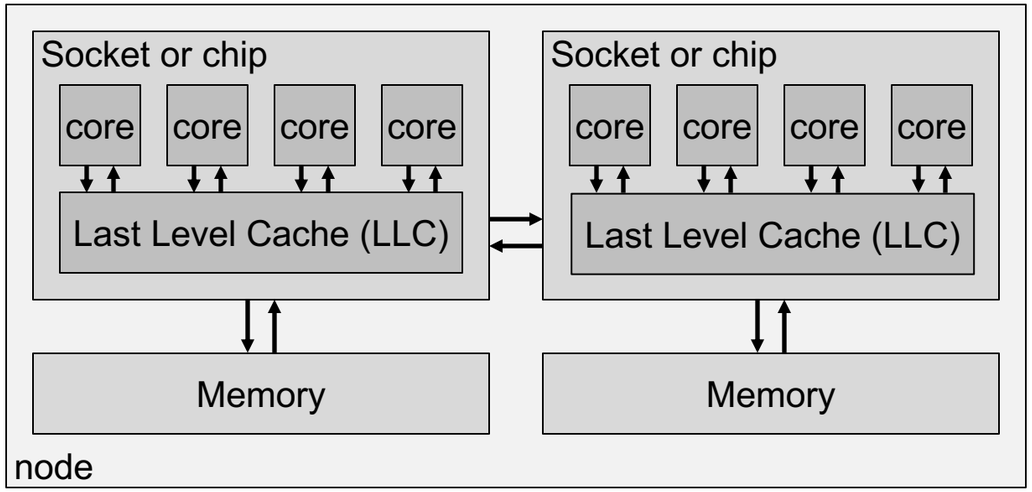
Acknowledgements

- 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.

Parallelization

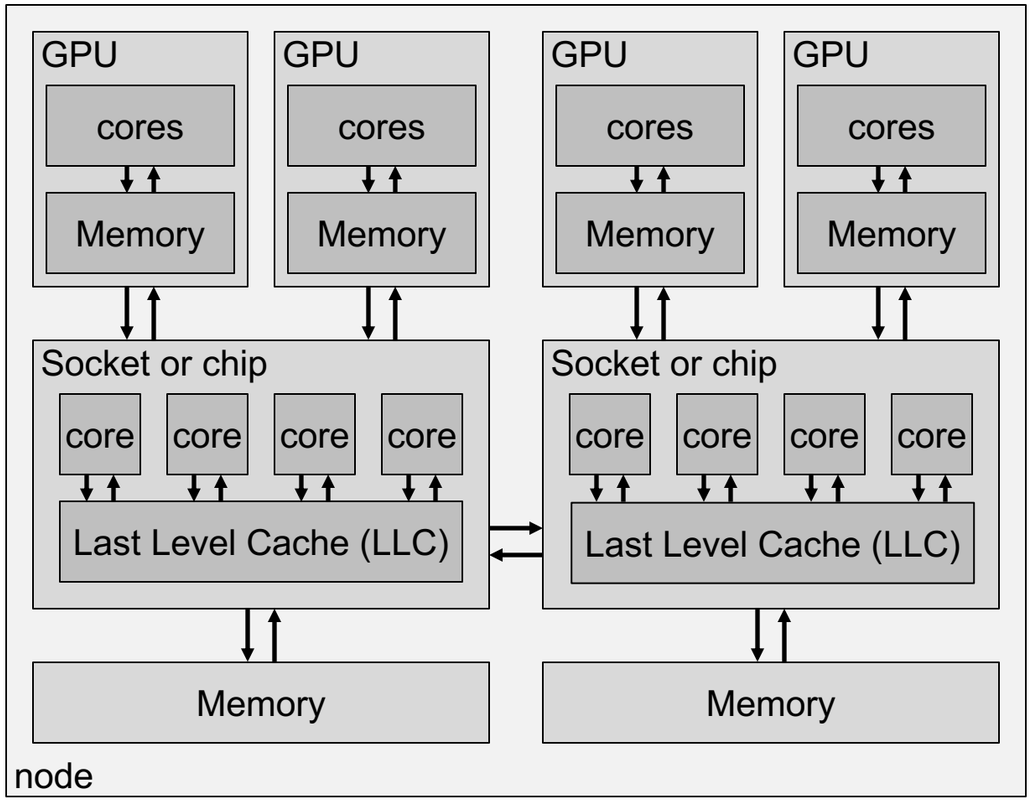
Multicore Node Architectures

- Imagine a dual-socket, quad-core node...
  - Each socket has a shared last level cache and its own directly attached memory
  - Between sockets, there is an interconnect (e.g. QPI, HT, etc…) that allows one socket to read another's memory/caches



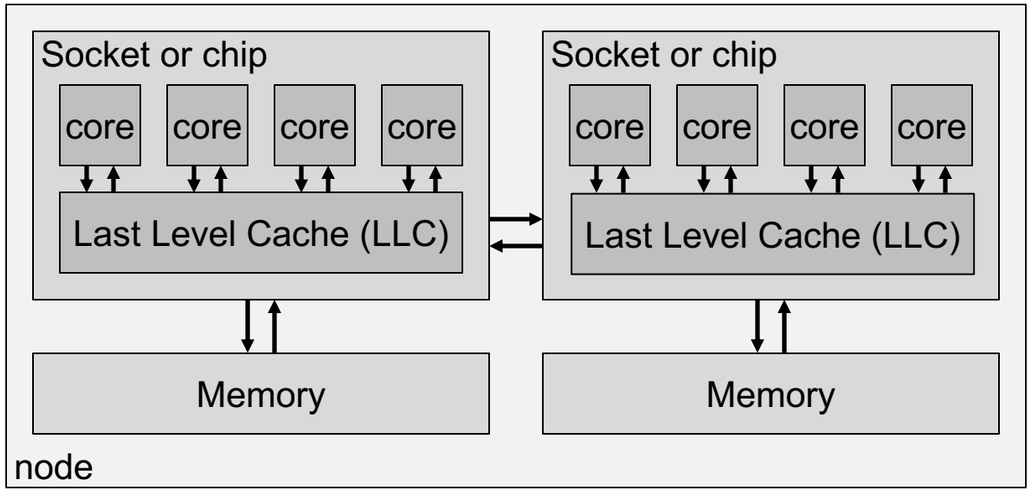
GPU-Accelerated Nodes

- Imagine a dual-socket, quad-core node...
  - Each socket has a shared last level cache and its own directly attached memory
  - Between sockets, there is an interconnect (e.g. QPI, HT, etc…) that allows one socket to read another's memory/caches
- Additionally, we may attach multiple GPUs per socket…
  - Each GPU has its own cores, caches, and memory.
  - Depending on the GPU, it can directly read host memory or other GPUs memory via loads/stores



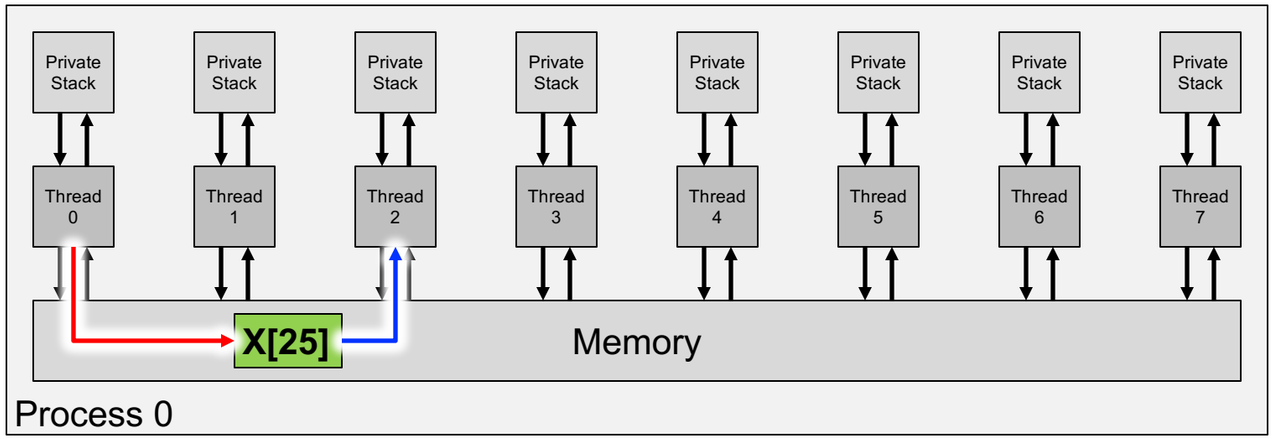
On-Node Parallelization

- We can apply a shared memory programming model (OpenMP/pthreads) and abstract away hardware…



On-Node Parallelization

- We can apply a shared memory programming model (OpenMP/pthreads) and abstract away hardware…
- As such, from SW POV, all we might see is threads and memory (no caches, sockets, cores)
- Threads read/write memory locations to exchange data

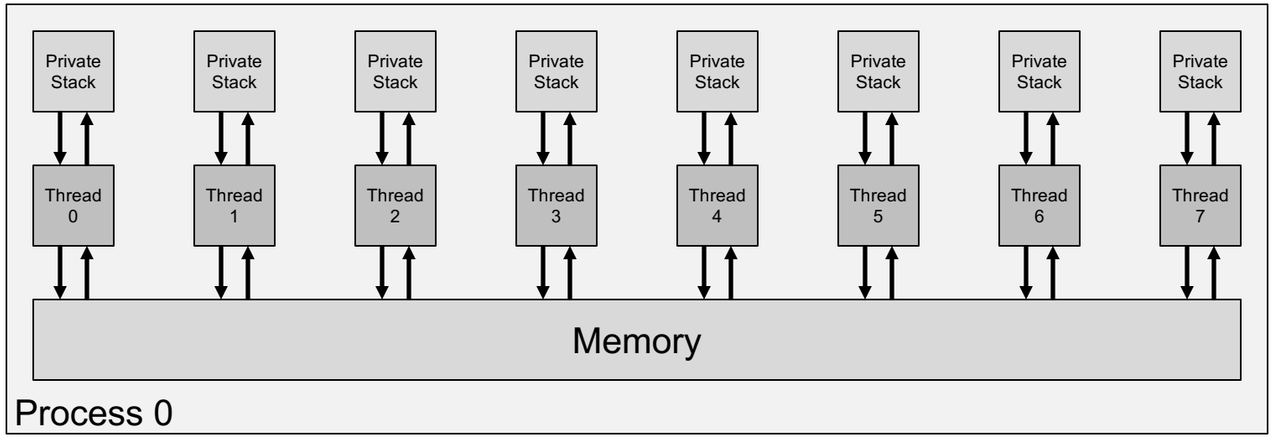


Thread0
X[25] = 3.14;

Thread 2
…
temp = X[25] * Y[27]

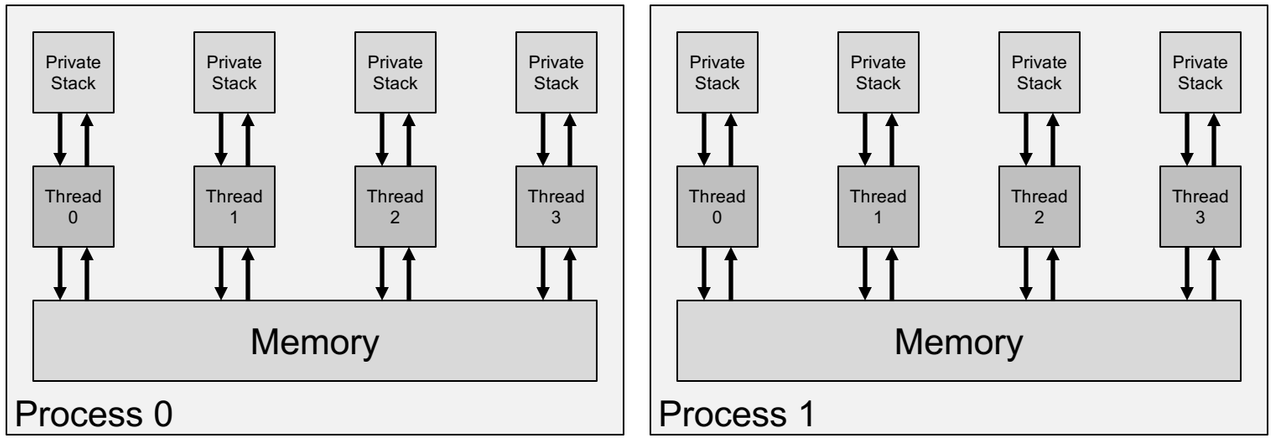
On-Node Parallelization

- Rather than only using threads, we can combine programming models (**Hybrid**)



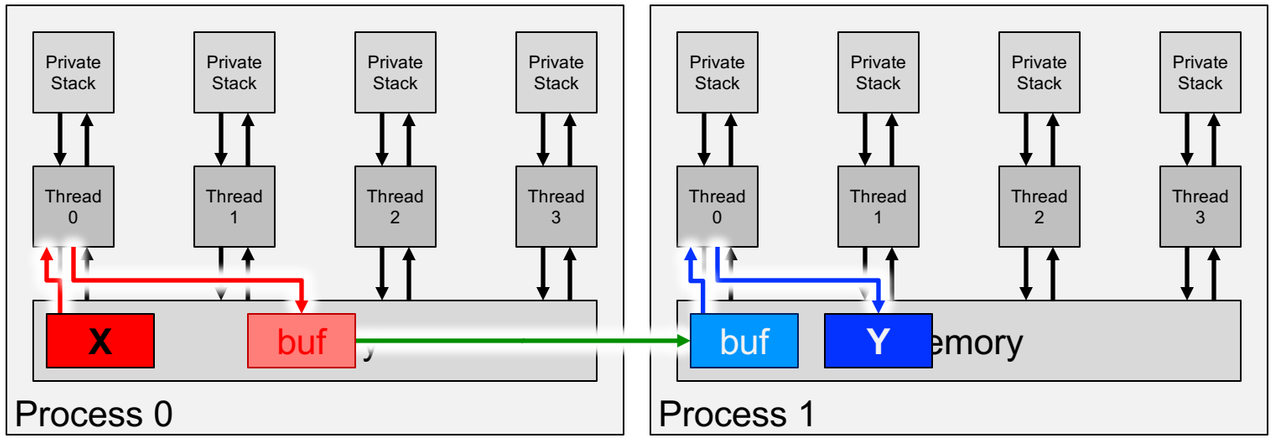
On-Node Parallelization

- Rather than only using threads, we can combine programming models (**Hybrid**)
- For example, we can combine MPI w/OpenMP…
  - 2 processes per node
  - 4 threads per process
  - With proper control of **affinity (srun and omp)**, we can efficiently map this to hardware.
  - Each process has its own private memory that cannot be read by the other process



On-Node Parallelization

- Communication among threads within a process is handled via shared memory
- Processes send/receive messages to exchange data
  - Messages are tagged by rank, ID, and communicator
  - Recv's on the destination can be posted before a Send has been posted on the host (blocks)
  - Asynchronous communication (isend/irecv) is non-blocking but requires a wait to ensure completion.

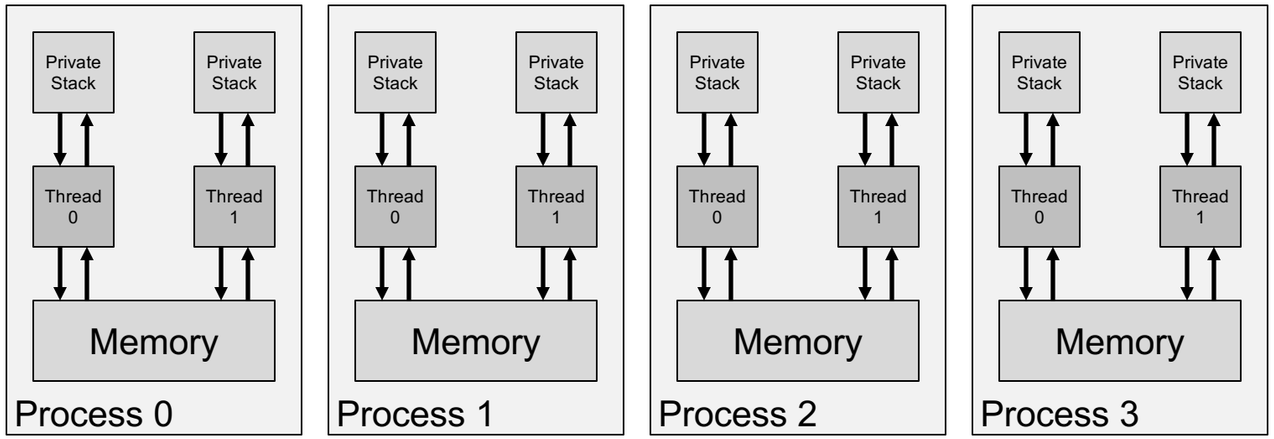


Process 0
Copy X to buf
Send buf to P2

Process 1
Recv buf from P0
…
Copy buf to Y

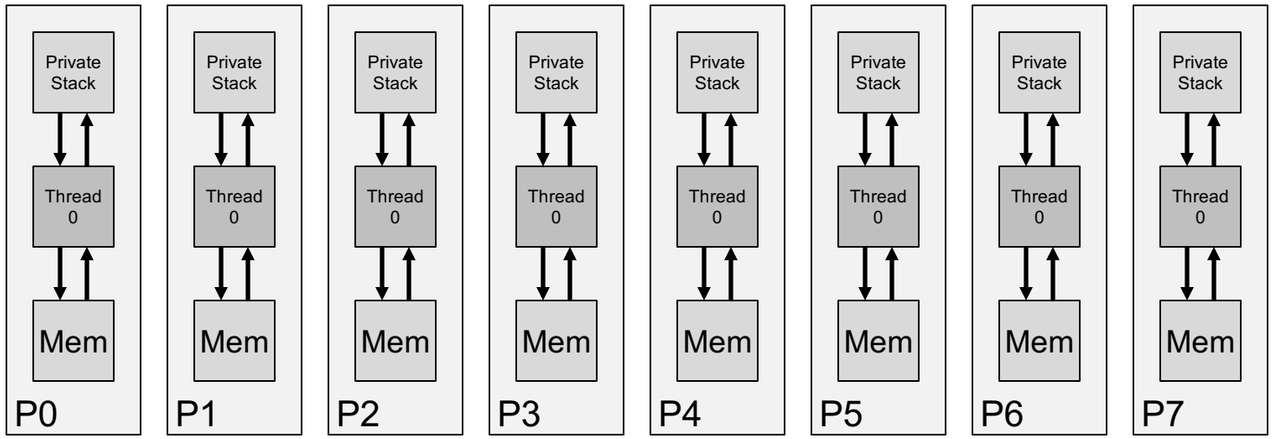
On-Node Parallelization

- We can select and combination of processes and threads on a node…
  - Its best not to oversubscribe hardware (procs * threads == cores)
  - Always use affinity to bind processes and threads cognizant of the underlying hardware.



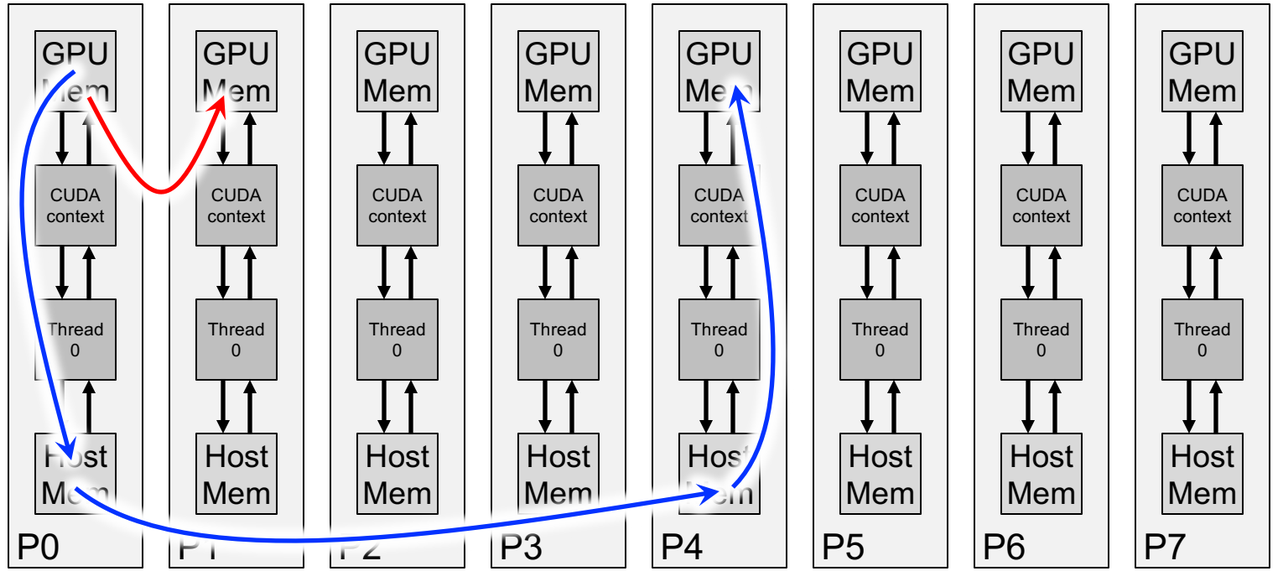
On-Node Parallelization

- We can select and combination of processes and threads on a node…
  - Its best not to oversubscribe hardware (procs * threads == cores)
  - Always use affinity to bind processes and threads cognizant of the underlying hardware.
  - **Note, OMP_NUM_THREADS==1 is not the same as flat MPI (no –qopenmp)**



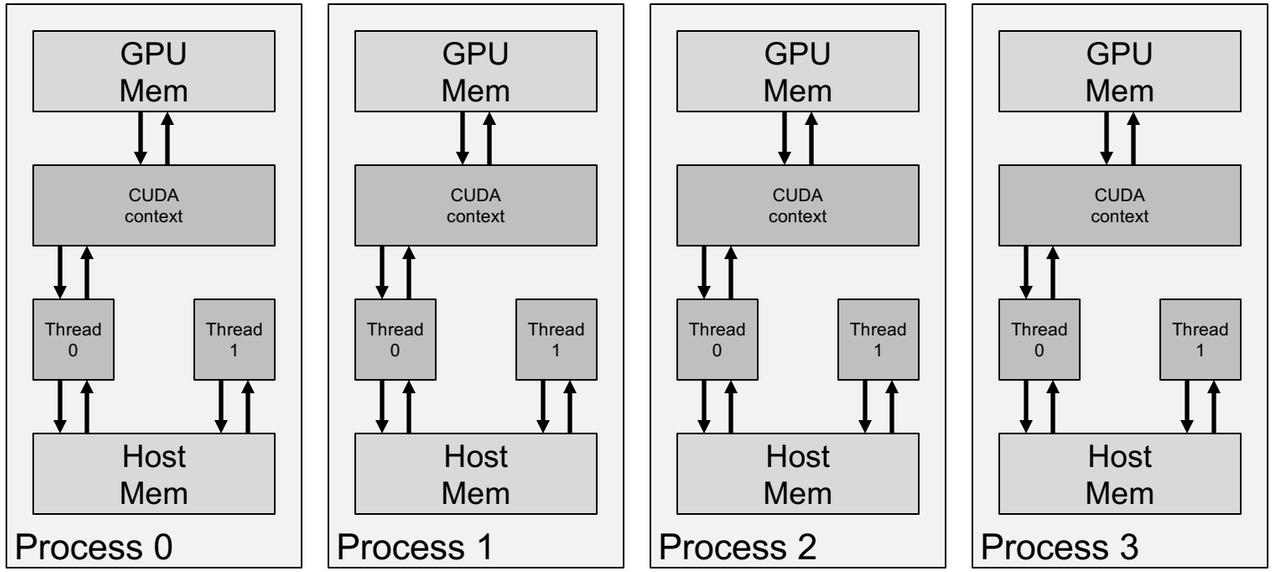
On-Node Parallelization

- Newer GPUs allow multiple processes to share a single GPU
  - each gets a ‘context’
  - GPU memory is portioned among contexts
  - Processes either time multiplex(older) resources or run concurrently
- Efficiency of GPU-GPU MPI exchanges is implementation dependent



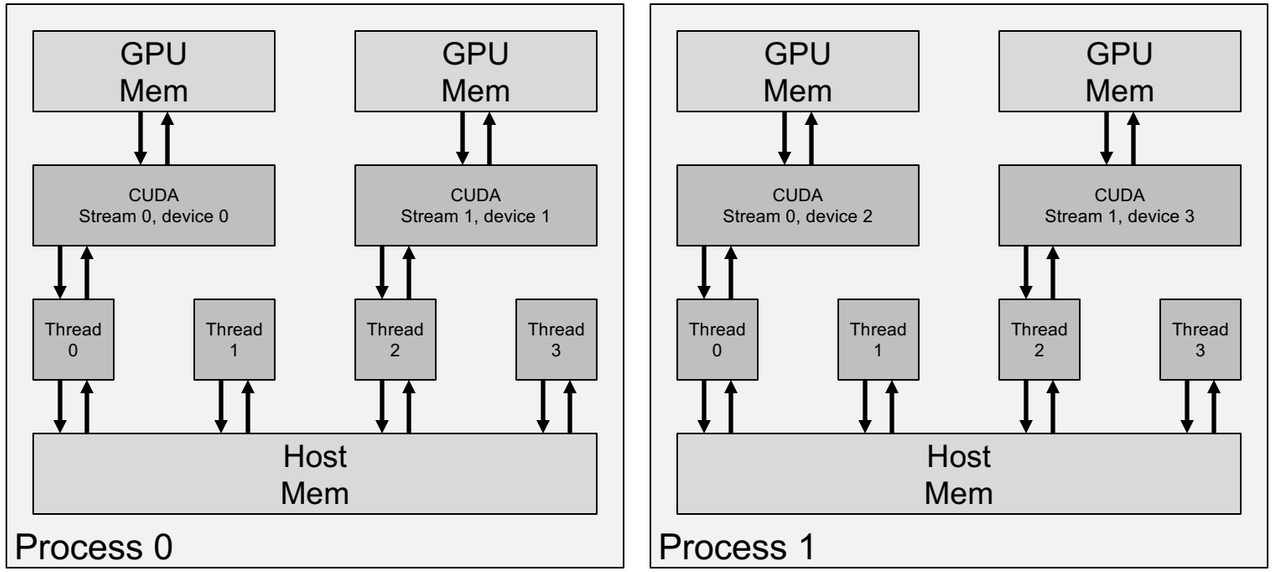
On-Node Parallelization

- One process per CPU core may not efficiently utilize a GPU-accelerated system
- As such, one could run one process per GPU…



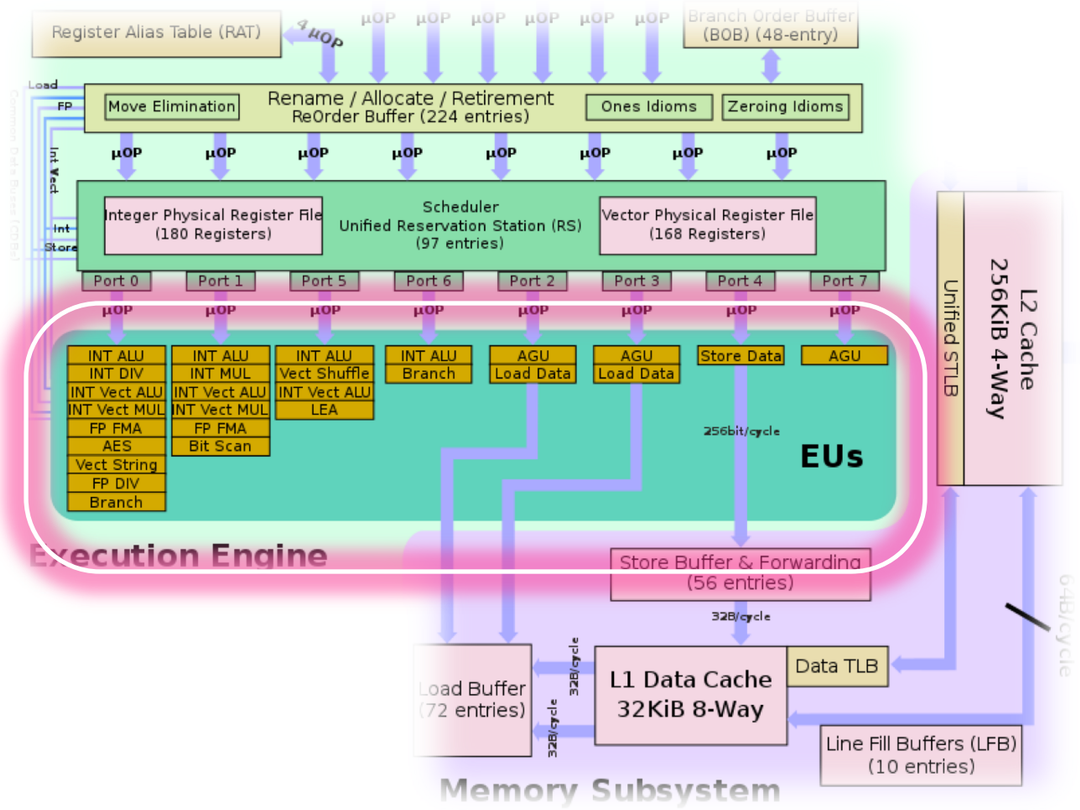
On-Node Parallelization

- As such, one process per CPU core may not efficiently utilize a GPU-accelerated system
- As such, one could run one process per GPU…
- … or one process per socket.
- Unfortunately, having one process control multiple GPUs can be harder to program (multiple streams)



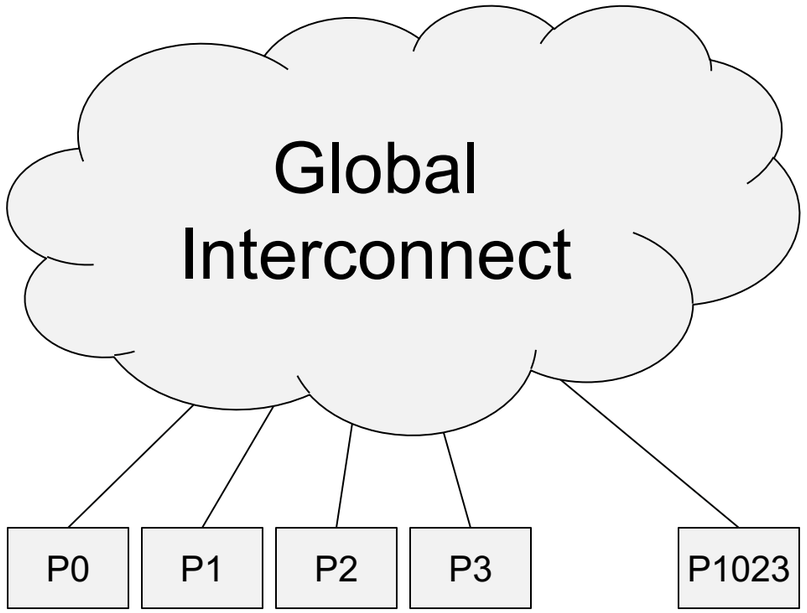
Intra-Core Parallelization

- Within each core there can be parallelism…
  - Multiple functional units (ILP)
  - Vectors/SIMD (DLP)
  - Specialized functional units (FMA, Tensor Cores, etc…)
  - Pipeline parallelism (ILP)
- Discovery can be done at compile time (vectors) or run time (ILP)
- Failure to exploit these limits performance



Multi-Node (Distributed Memory) Parallelization

- MPI is used for multi-node parallelization
  - From the user standpoint, on-node and inter-node MPI looks the same
  - All communication is handled via MPI (P2P or collectives)… no shared memory communication
- UPC, CAF, GA provide shared memory abstraction.
- Memory balancing…
  - can't shift memory capacity from one node to another.
  - If any one node exceeds its memory capacity, the job will crash w/OOM error
  - Swap is rare (diskless compute nodes)



Synchronization

- When one thread (or process) produces data and another thread (or process) consumes it, we have a data hazard
- We must synchronize to resolve the data hazard and ensure determinism & correctness.

Synchronization

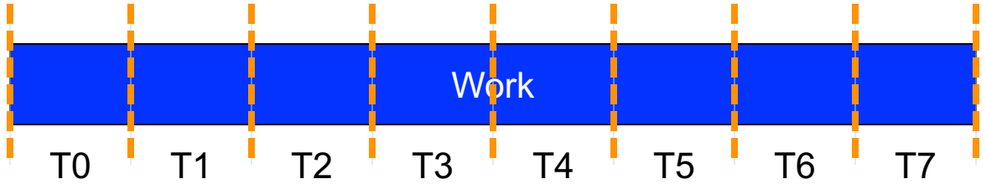
- In the P2P MPI world, synchronization is automatic in the irecv/isend/waitall/compute paradigm

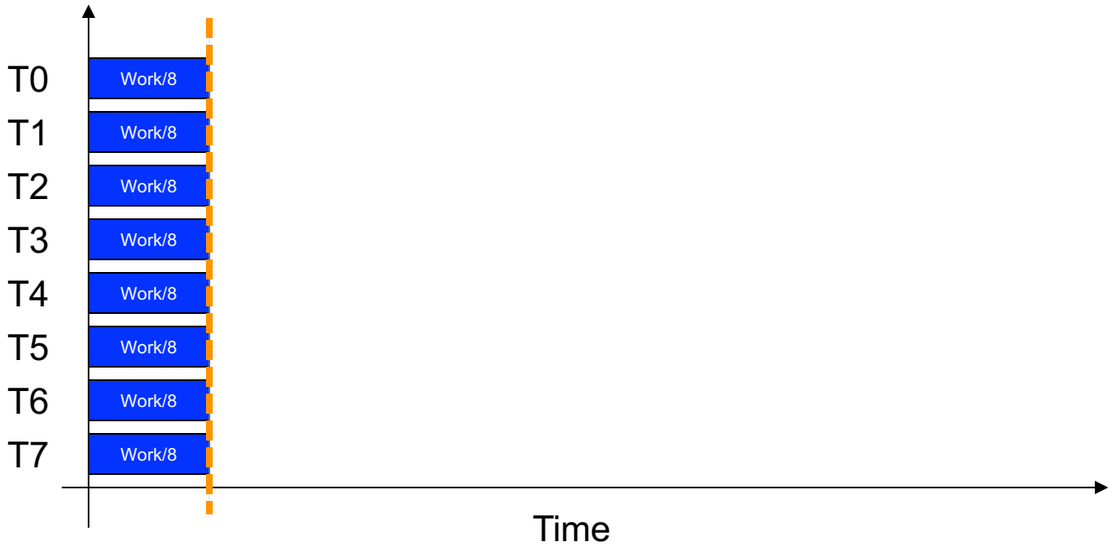
- In the OpenMP world, we must explicitly synchronize…
- Easiest solution: leverage BSP model…
  - #pragma omp parallel implies an implicit barrier
  - Data is only exchanged between parallel regions (never within)
  - Coarse grained
- Finer grained…
  - Use OMP reductions, atomics, locks, and critical sections

Challenges

Load Balancing

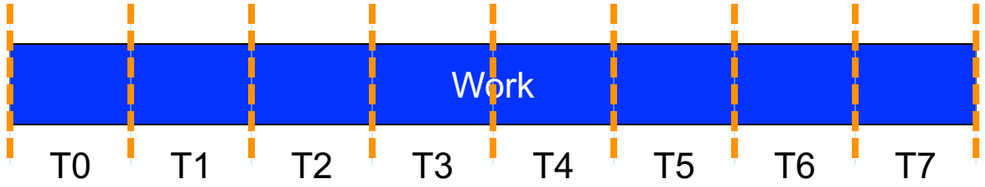
- Imagine we have some work to partition (e.g. computation on an array)
  - We can try and uniformly partition work (loop iterations) among threads
  - Ideally, the run time should be reduced by 1/NThreads

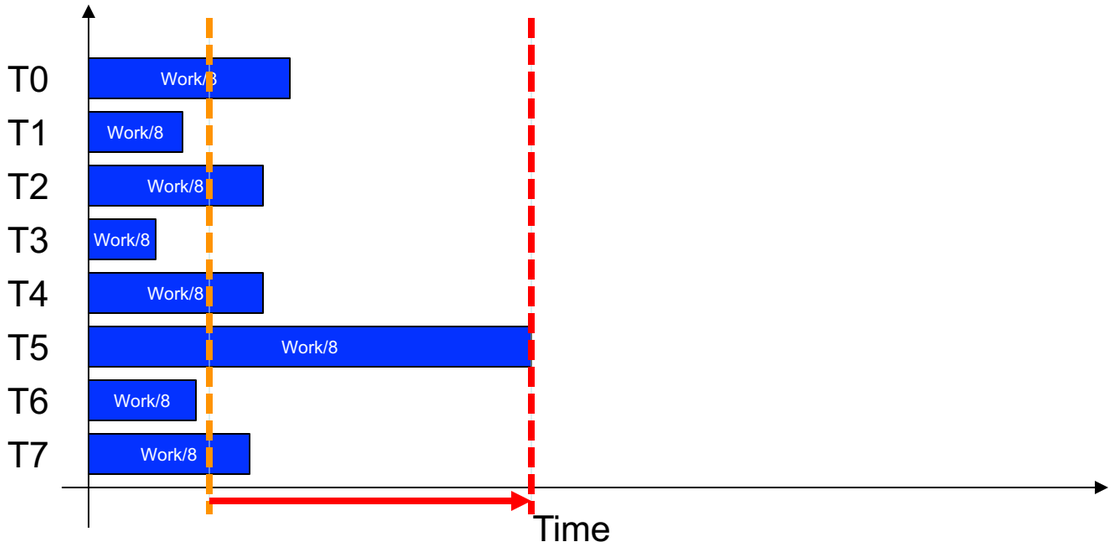




Load Balancing

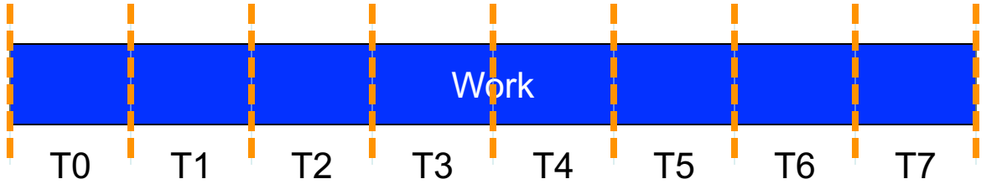
- Unfortunately, some loop iterations may be more expensive, or some threads may run slower (e.g. cache effects)
  - As a result, we can observe load imbalance where run time is limited by the slowest thread
  - We can assess the degree of load imbalance by measuring **max/average**
  - **A slow outlier may substantially hurt performance**, but…

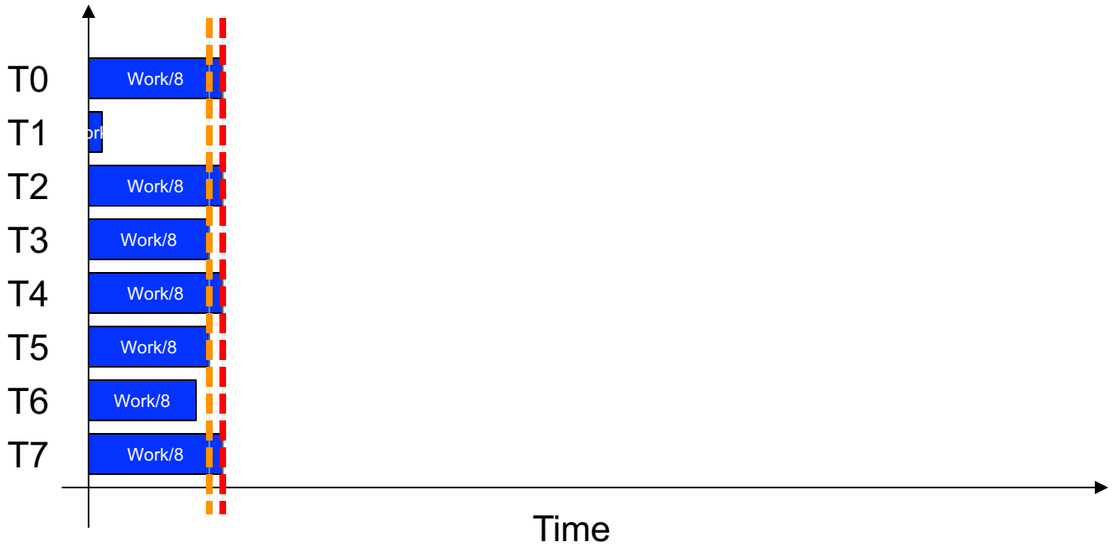




Load Balancing

- Unfortunately, some loop iterations may be more expensive, or some threads may run slower (e.g. cache effects)
  - As a result, we can observe load imbalance where run time is limited by the slowest thread
  - We can assess the degree of load imbalance by measuring **max/average**…
  - **A slow outlier may substantially hurt performance**, but…
  - …a fast thread may not help or hurt much





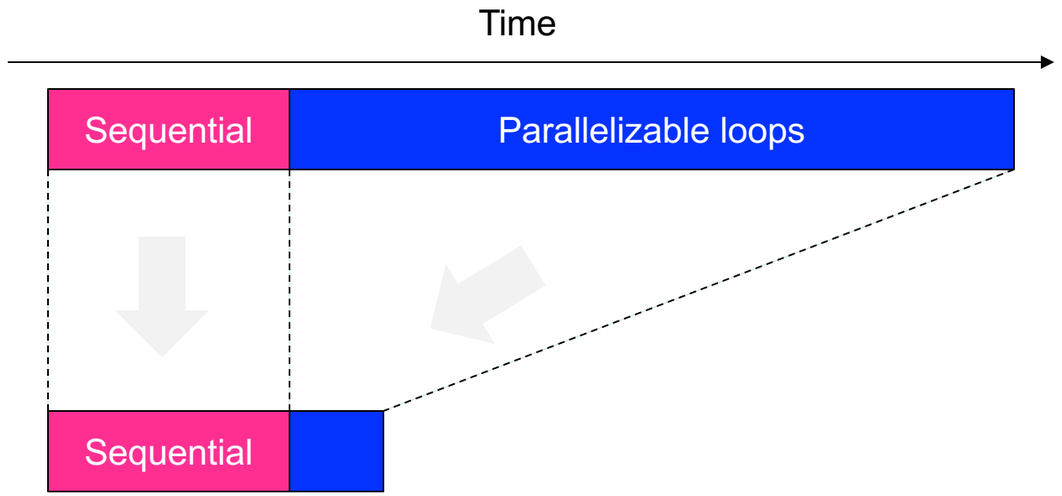
Lack of Parallelism

- Trends in architecture have enabled >>1000-way parallelism on a chip (#FPUs * FPU latency)
- Not all loop nests support 1000-way parallelization
- Loop nests with <1000-way parallelism underutilize HW resources

- Often codes must be restructured to enable more parallelism
  - Loops are reordered/fused (OMP collapse(3))
  - Variables(arrays) are privatized and reduced
  - Nominally sequential functions/solvers on independent variables are performed concurrently (MPI sub communicators or OMP Tasks)
  - Workflows/multiphysics are parallelized at launch (SLURM MPMD)

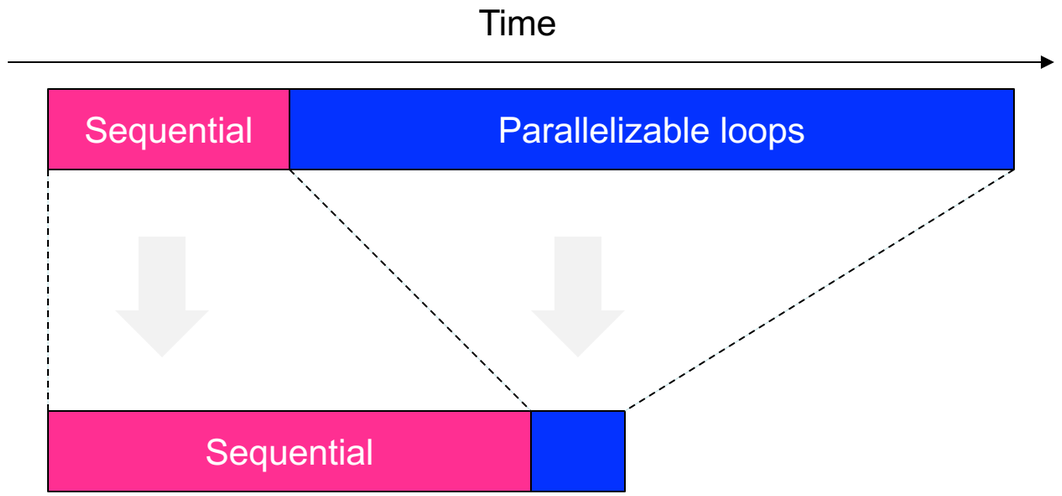
Amdahl's Law & Bottlenecks

- In an application, not every loop might be parallelized.
  - Speedup = $(f_{seq} + f_{par}/S_{par})^{-1}$
  - 8x speedup on 75% of the run time provides a 3x speedup
  - **Infinite** speedup on 75% of the run time only provides a **4x speedup**.
- Similar effect applies to vectors, GPUs, and other accelerators.
  - e.g. GPU providing 10x on 10% of an application = **10% speedup**.



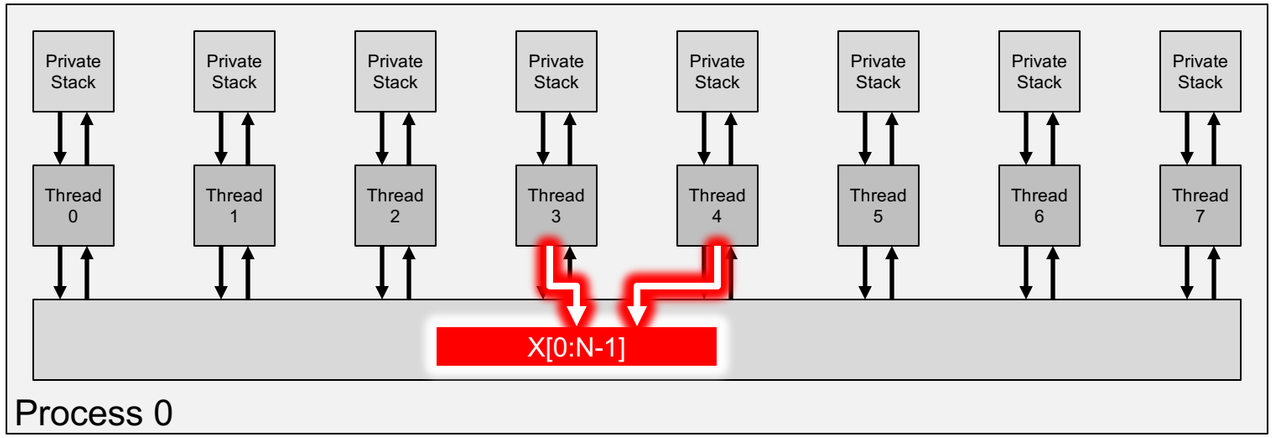
Amdahl's Law & Bottlenecks

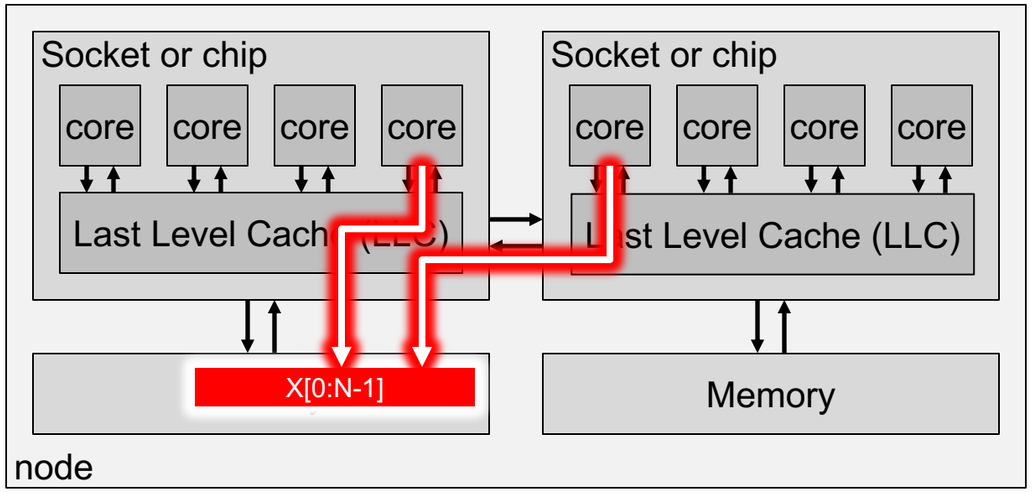
- On manycore processors (KNL), single thread performance is worse than Haswell
  - Parallel parts of apps get faster
  - Sequential parts get slower
  - **Overall benefit is less than expected**



Non-Uniform Memory Access (NUMA)

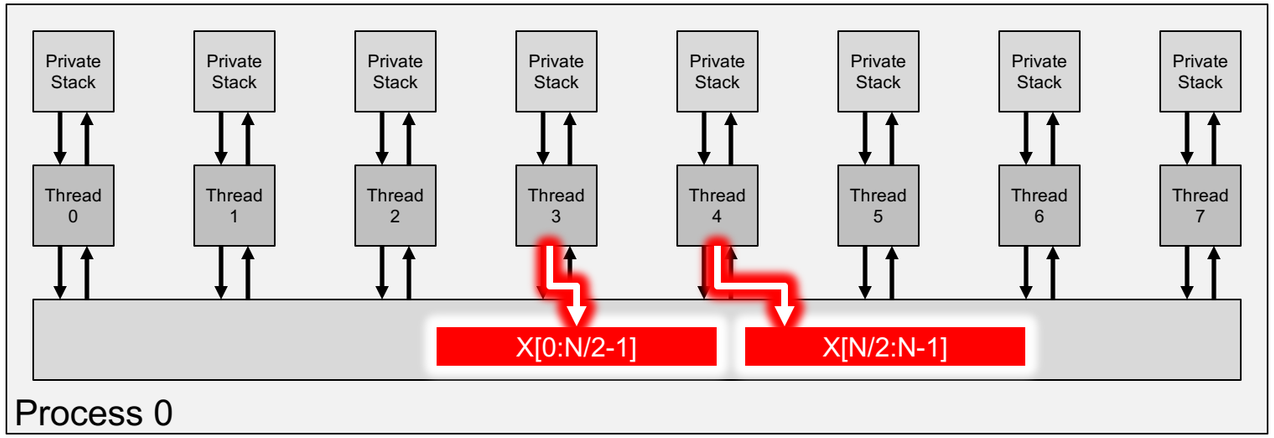
- In a shared memory, any thread can read any memory location
- However, memory bandwidth and latency can vary (NUMA)
  - Consider array double X[N];
  - If naively allocated, all data is placed on socket 0
  - Latency/bandwidth to element x[i] is different for thread 3 and thread 4
  - Socket 0's memory bandwidth is highly contended while socket 1's is underutilized

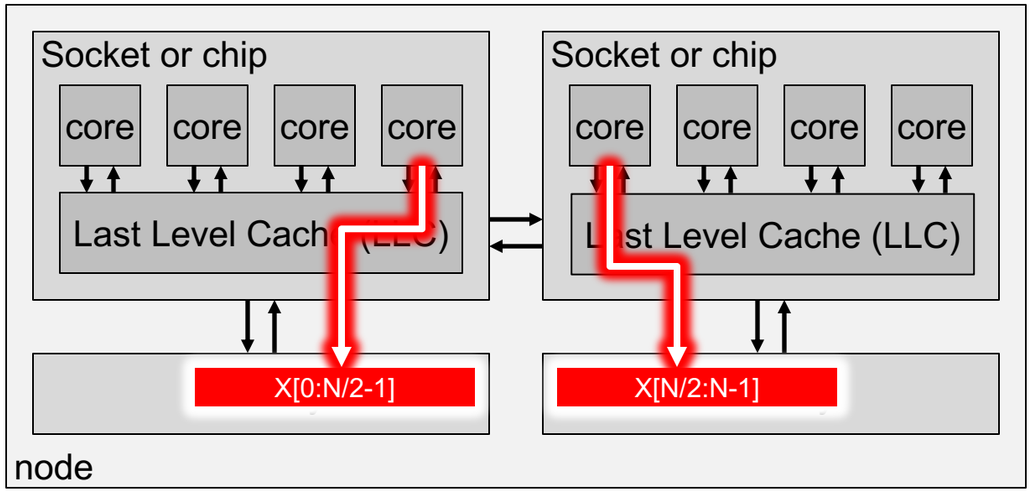




Non-Uniform Memory Access (NUMA)

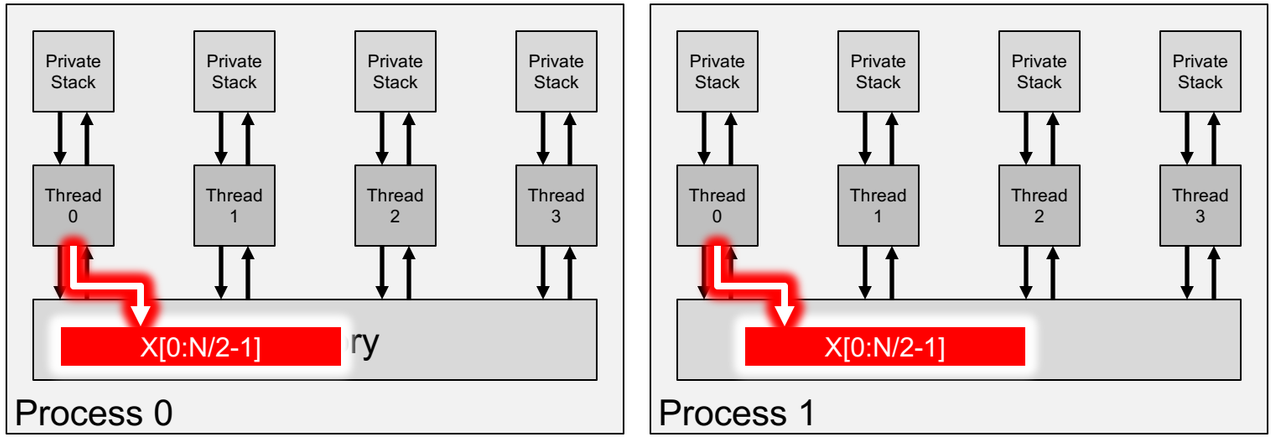
- Allocate data based on how we intend to access it
- Rectify this NUMA effect via ***first touch initialization***
  - Pages are allocated with affinity to the core that first touched the data
  - Parallelizing the initialization transparently partitions the array among NUMA nodes
  - n.b. This only works on the first allocation of this memory page (subsequent free/malloc corrupt the process)
  - **Explicit thread affinity is imperative**

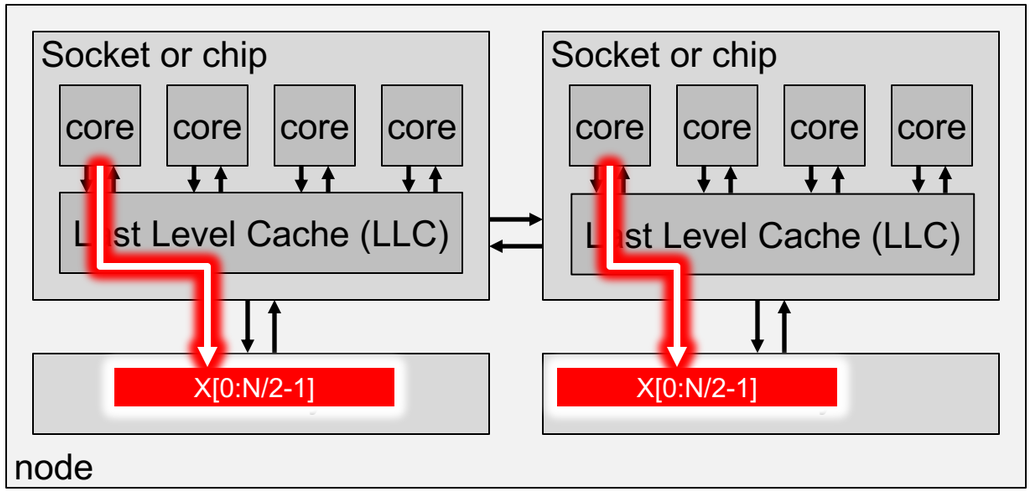




Non-Uniform Memory Access (NUMA)

- Simplest solution is to run at least **one process per NUMA node**
  - All data for that process is allocated on the numa node with affinity to its cores
  - All data access are restricted to that NUMA node





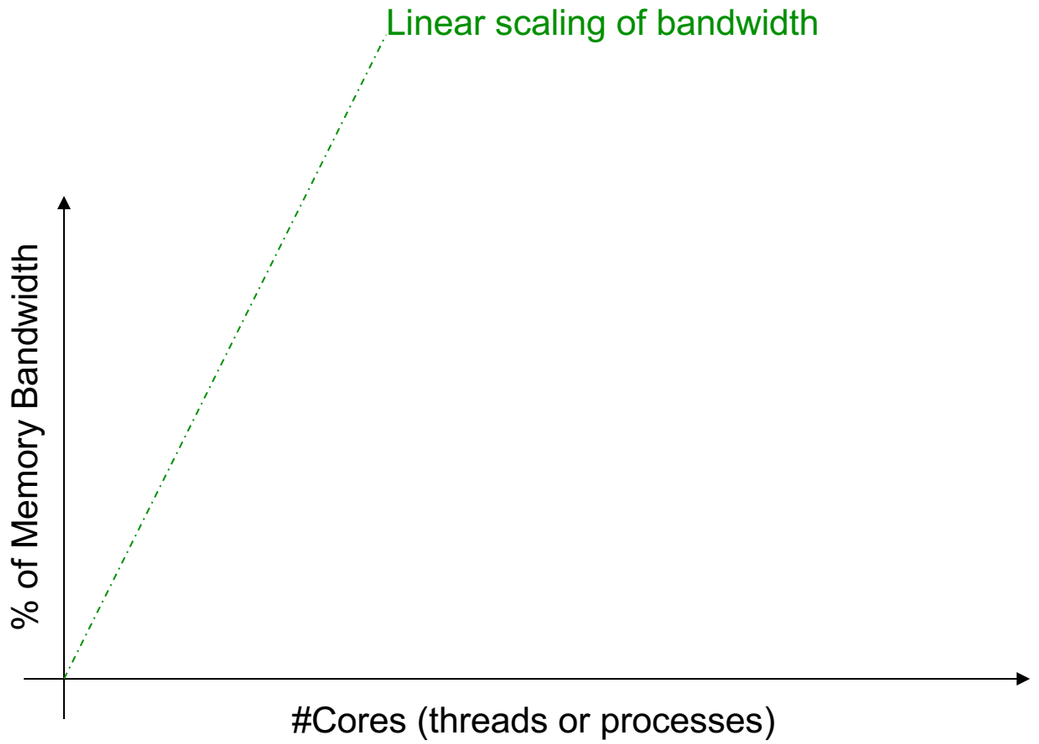
Memory/Cache Capacity Contention

- Each thread or process exerts some pressure on the cache (on-chip L1/L2 or off-chip MCDRAM/HBM memory).
- As thread/process-concurrency increases, the requisite active working set **can exceed cache capacity**
  - When this happens, capacity misses manifest, data movement increases (even if perfectly computationally load balanced), and **performance can plummet**.
  - If this happens in MCDRAM or HBM caches, we run at DDR speeds or Nvlink speeds.

- To rectify this, we often **tile** or **block** loops so that the per-thread working set remains sufficiently small.

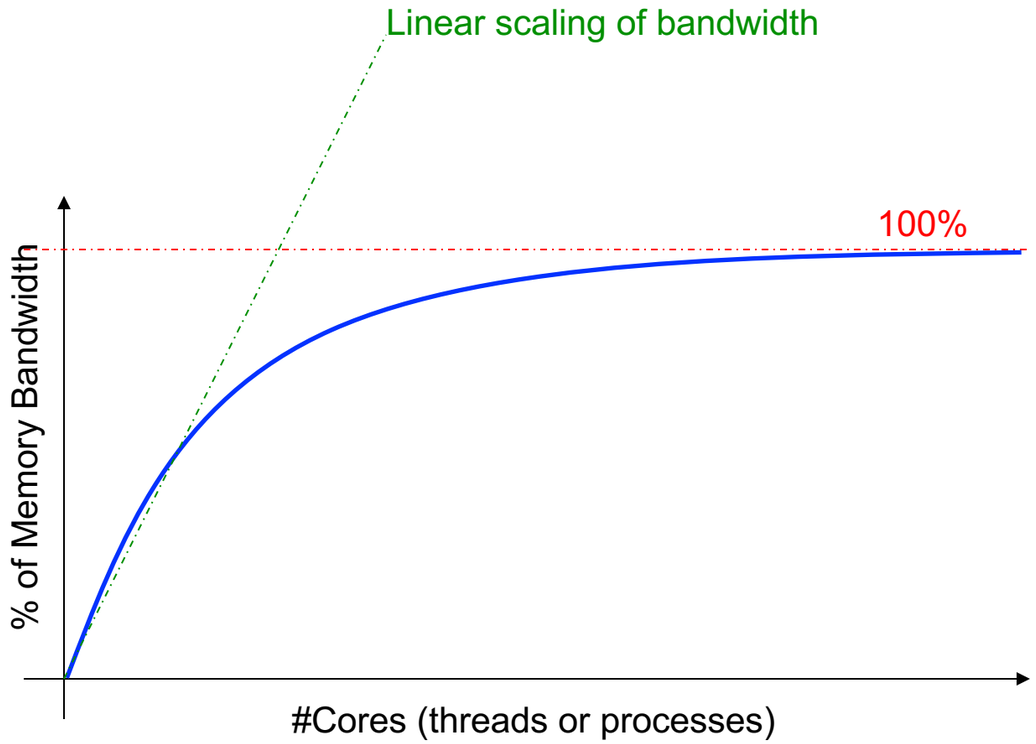
Memory/Cache Bandwidth Contention

- There is less memory bandwidth than all cores could consume
  - 1 core can drive >10GB/s of memory bandwidth
  - 16 cores (e.g. 1P HSW) could drive >160



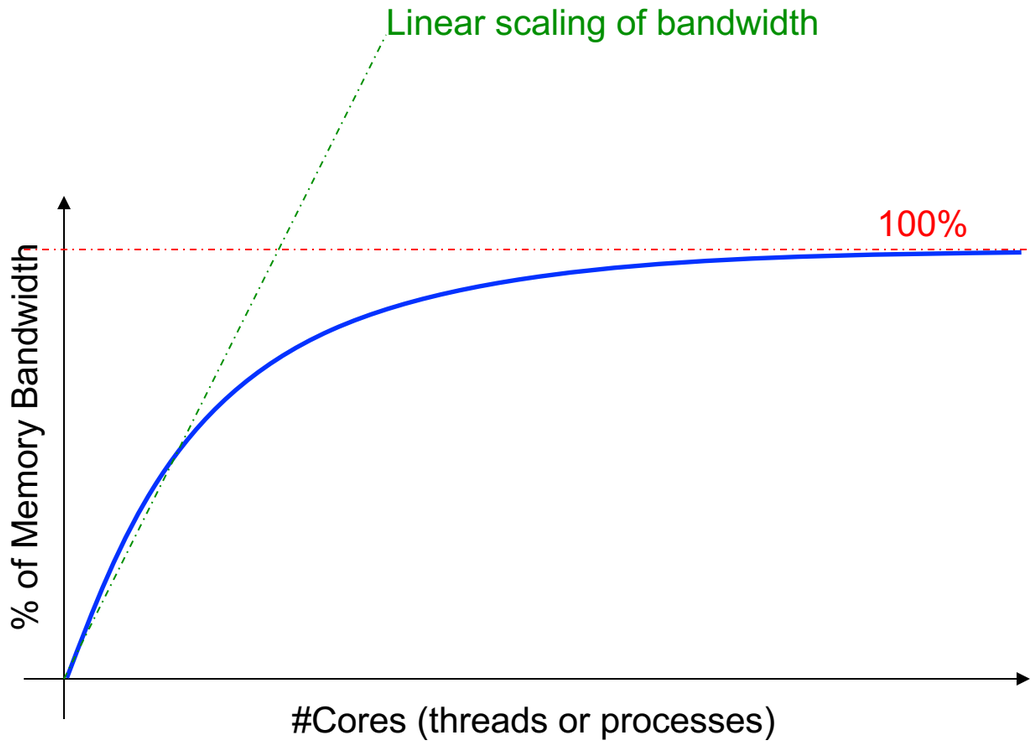
Memory/Cache Bandwidth Contention

- There is less memory bandwidth than all cores could consume
  - 1 core can drive >10GB/s of memory bandwidth
  - 16 cores (e.g. 1P HSW) could drive >160
  - Socket only has 50GB/s available.
  - **memory bandwidth has become a bottleneck**
  - A similar effect can emerge on LLC or MCDRAM caches



Memory/Cache Bandwidth Contention

- There is less memory bandwidth than all cores could consume
  - 1 core can drive >10GB/s of memory bandwidth
  - 16 cores (e.g. 1P HSW) could drive >160
  - Socket only has 50GB/s available.
  - **memory bandwidth has become a bottleneck**
  - A similar effect can emerge on LLC or MCDRAM caches
- "Flops are Free"
  - **bottleneck or opportunity?**



Network Performance

- Each message sent via MPI incurs some non-zero overhead (and latency)
- We can proxy message time as:

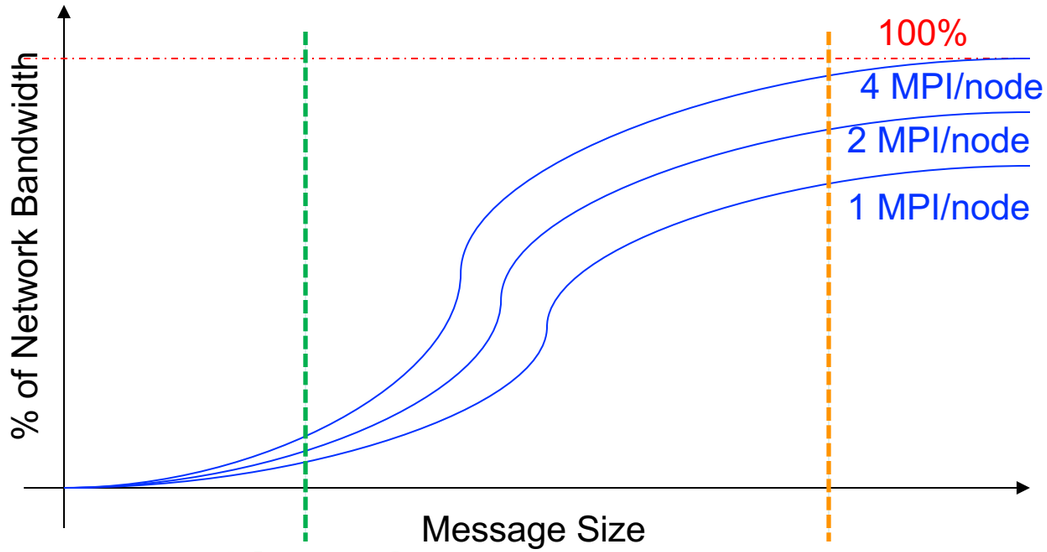
  time = overhead + size/bandwidth

- As such, network utilization…

  = size / (overhead*bandwidth + size)

  = (overhead*bandwidth/size + 1)$^{-1}$

Latency-Bandwidth Product



Need lots of processes/node if typical message size is small

Need relatively few processes/node if typical message size is large

Network Performance

- Each message sent via MPI incurs some non-zero overhead (and latency)
- We can proxy message time as:

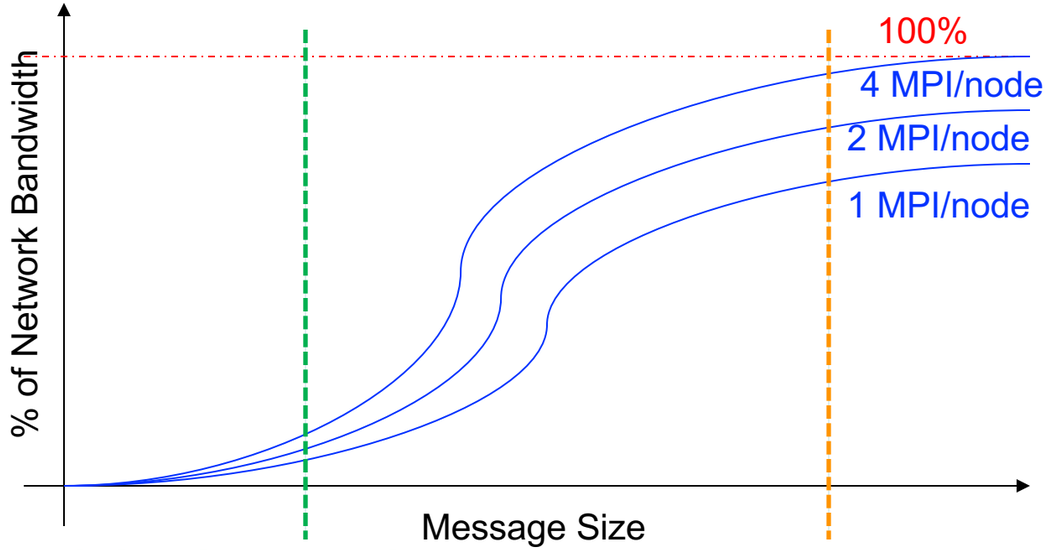
  time = overhead + size/bandwidth

- As such, network utilization…

  = size / (overhead*bandwidth + size)

  = (overhead*bandwidth/size + 1)$^{-1}$

Latency-Bandwidth Product



Incentivized to send fewer messages (even if bigger)

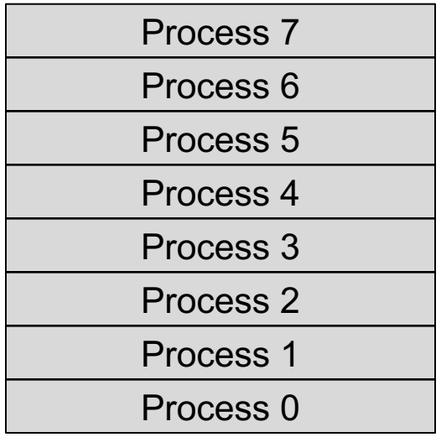
Incentivized to send less data (smaller messages)

Network Contention

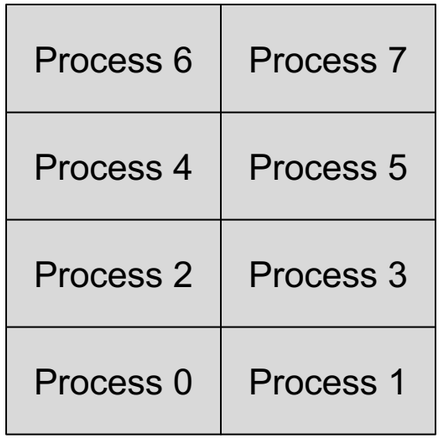
- We can saturate different aspects of the network…
- Injection Bandwidth
  - nodes cannot inject data faster into the network
  - e.g. broadcasts or large messages
- Ejection Bandwidth
  - node cannot eject data from the network fast enough
  - e.g. reductions (single node must receive data from all other nodes)
- Bisection Bandwidth
  - Bandwidth (links) connecting conceptual partitions of the network
  - Artifact of the topology (architecture) of the network
  - Job placement (and decomposition) can mitigate this

Surface:Volume

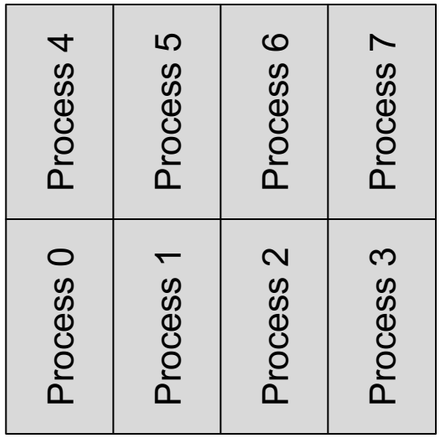
- When we decompose a problem among processes, we must do so in a manner that…
  - Minimizes the number of messages (recall overhead per message)
  - Minimizes the surface:volume ratio (i.e. ratio of MPI data movement : local flops)
- Consider 8-way MPI parallelization of a PDE on a NxN structured grid…



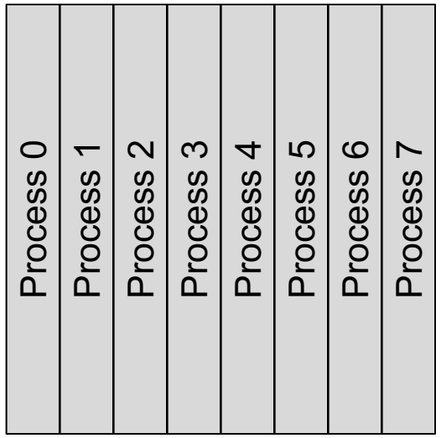
S:V = 2.25N / N^2



S:V = 1.5N / N^2



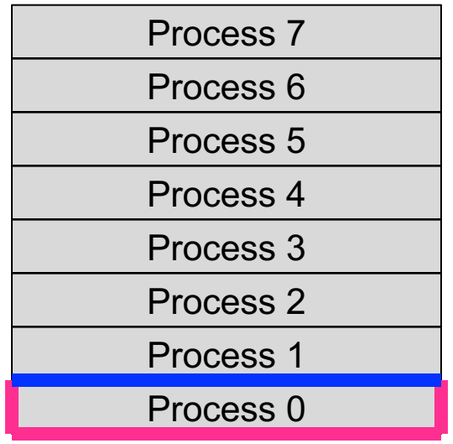
S:V = 1.5N / N^2



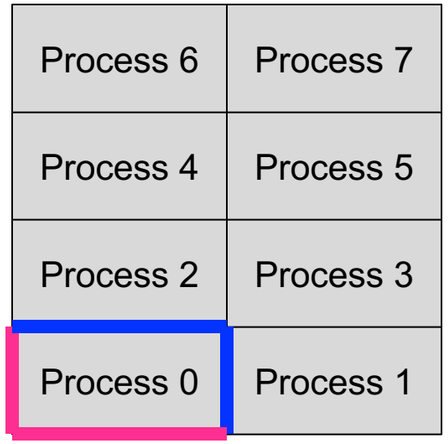
S:V = 2.25N / N^2

Surface:Volume

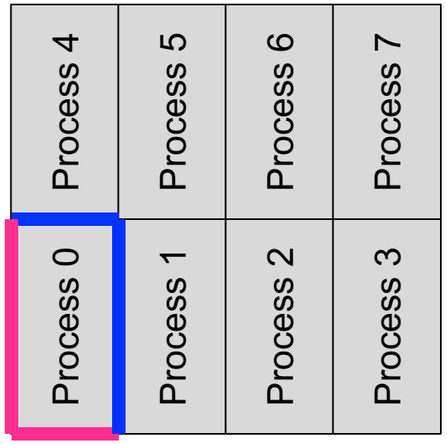
- On a multicore node, think about how much data is **off node exchanges** vs. **on-node exchanges.**
- On-node exchanges should be faster (DRAM BW >> Network BW)
- Different off-node surface:volume ratios leads to **load imbalance**.



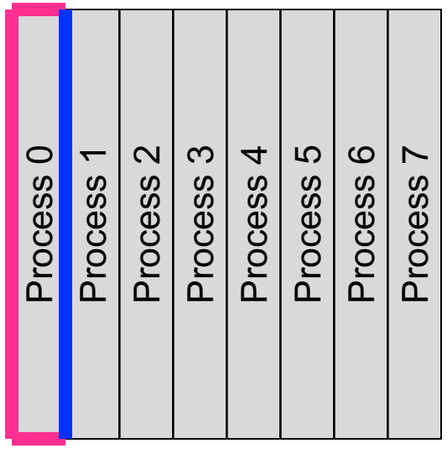
S:V = 1.25N / N^2



S:V = 0.75N / N^2



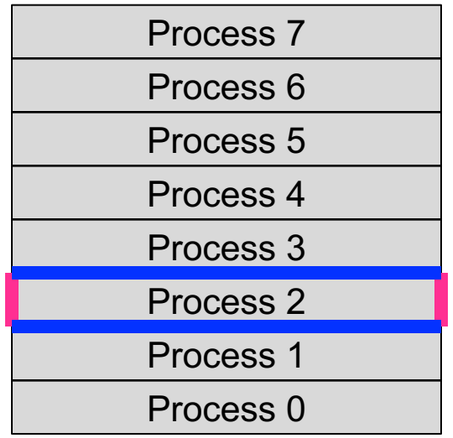
S:V = 0.75N / N^2



S:V = 1.25N / N^2

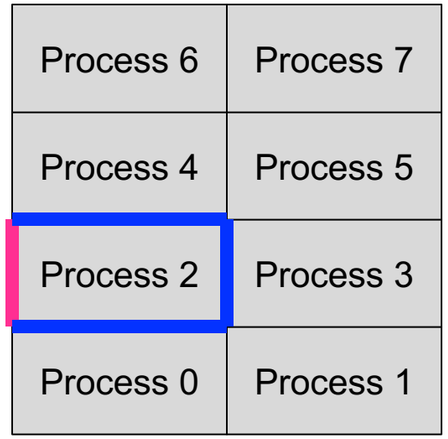
Surface:Volume

- On a multicore node, think about how much data is **off node exchanges** vs. **on-node exchanges.**
- On-node exchanges should be faster (DRAM BW >> Network BW)
- Different off-node surface:volume ratios leads to **load imbalance**.



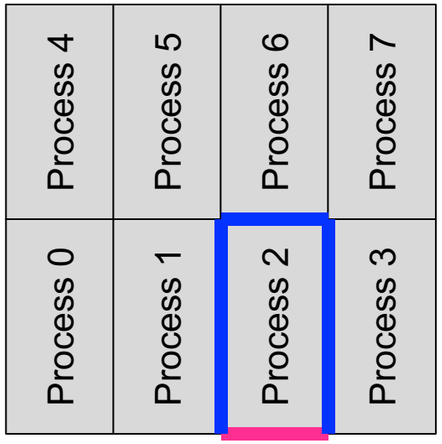
S:V = 1.25N / N^2

S:V = 0.25N / N^2



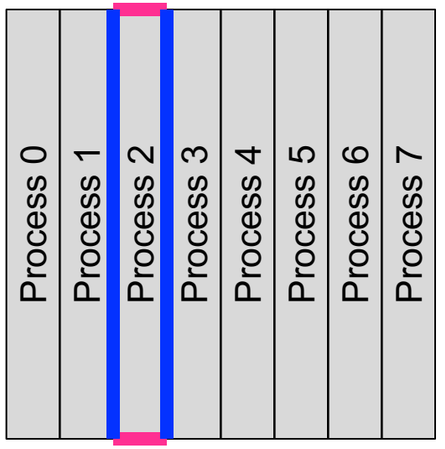
S:V = 0.75N / N^2

S:V = 0.25N / N^2



S:V = 0.75N / N^2

S:V = 0.25N / N^2



S:V = 1.25N / N^2

S:V = 0.25N / N^2

Little's Law Redux…

- Recast latency-bandwidth product for OMP/CUDA overheads & flop/s...
- Haswell (Xeon CPU):
  - 100 GB/s, 1.3 Tflop/s, ~1us OMP overhead
  - **Can't hit peak bandwidth on any kernel that moves less than 100KB**
  - **Can't hit peak flops on any kernel that does less than 1M FP operations**
- KNL (Xeon Phi Manycore):
  - 400 GB/s, 2.5 Tflop/s, ~5us OMP overhead
  - **Can't hit peak bandwidth on any kernel that moves less than 2MB**
  - **Can't hit peak flops on any kernel that does less than 13M FP operations**
- Volta GPU:
  - 800 GB/s, 7 Tflop/s, ~20us CUDA launch overhead
  - **Can't hit peak bandwidth on any kernel that moves less than 16MB**
  - **Can't hit peak flops on any kernel that does less than 140M FP operations**

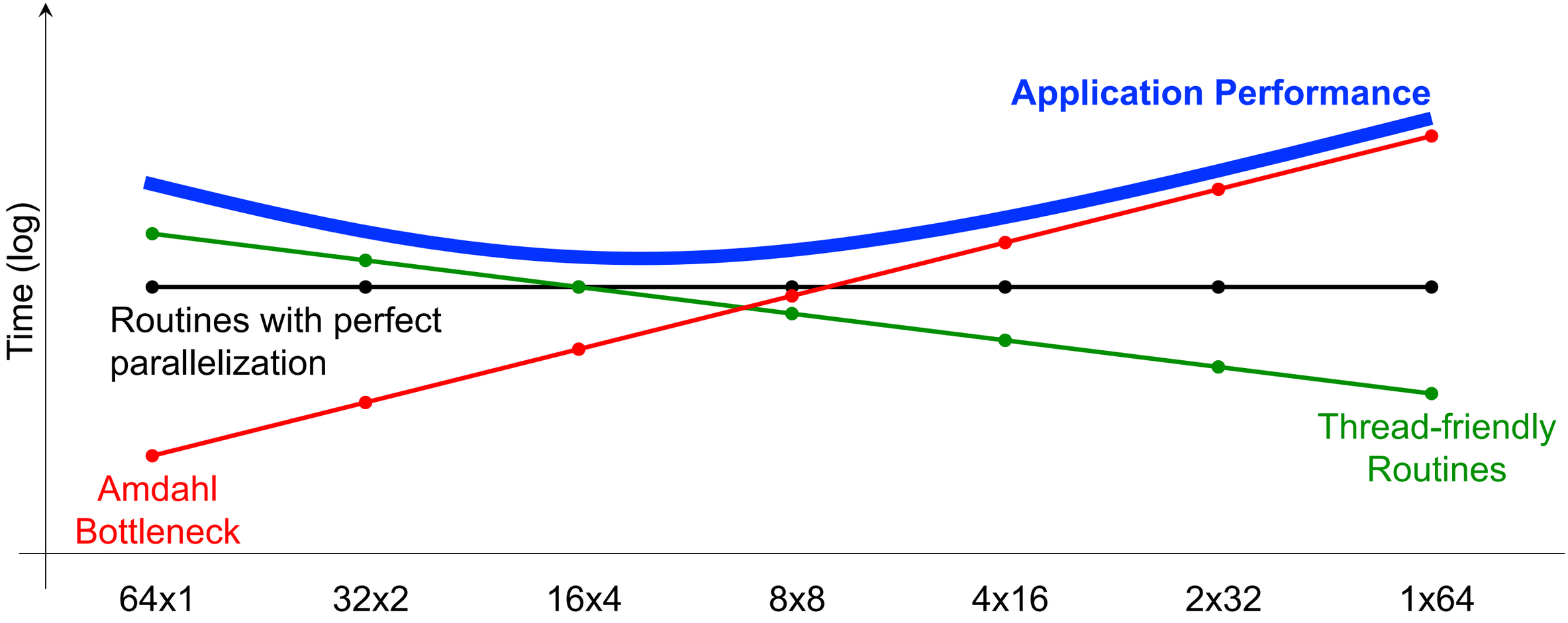
Trading Process Concurrency for Thread Concurrency

Process-Heavy *Thread-Heavy*

- ✓ *Single Programming Model (MPI)*
- ✓ *Maximizes MPI bandwidth*
- ✓ *Implicitly addresses affinity and NUMA*
- ✓ *Minimizes superfluous synchronization*
- ✓ *Eliminates Amdahl (threading) bottlenecks*

- ✓ *Minimizes MPI data movement*
- ✓ *Memory capacity friendly… avoids duplication of data*
- ✓ *Bandwidth friendly… access to shared data is handled via caches instead of duplication*
- ✓ *Compute-friendly… ideally avoids redundant computation*
- ✓ *Load balancing is simplified.*
- ✓ *Job placement is simplified.*

Trading Process Concurrency for Thread Concurrency



Performance and Scalability

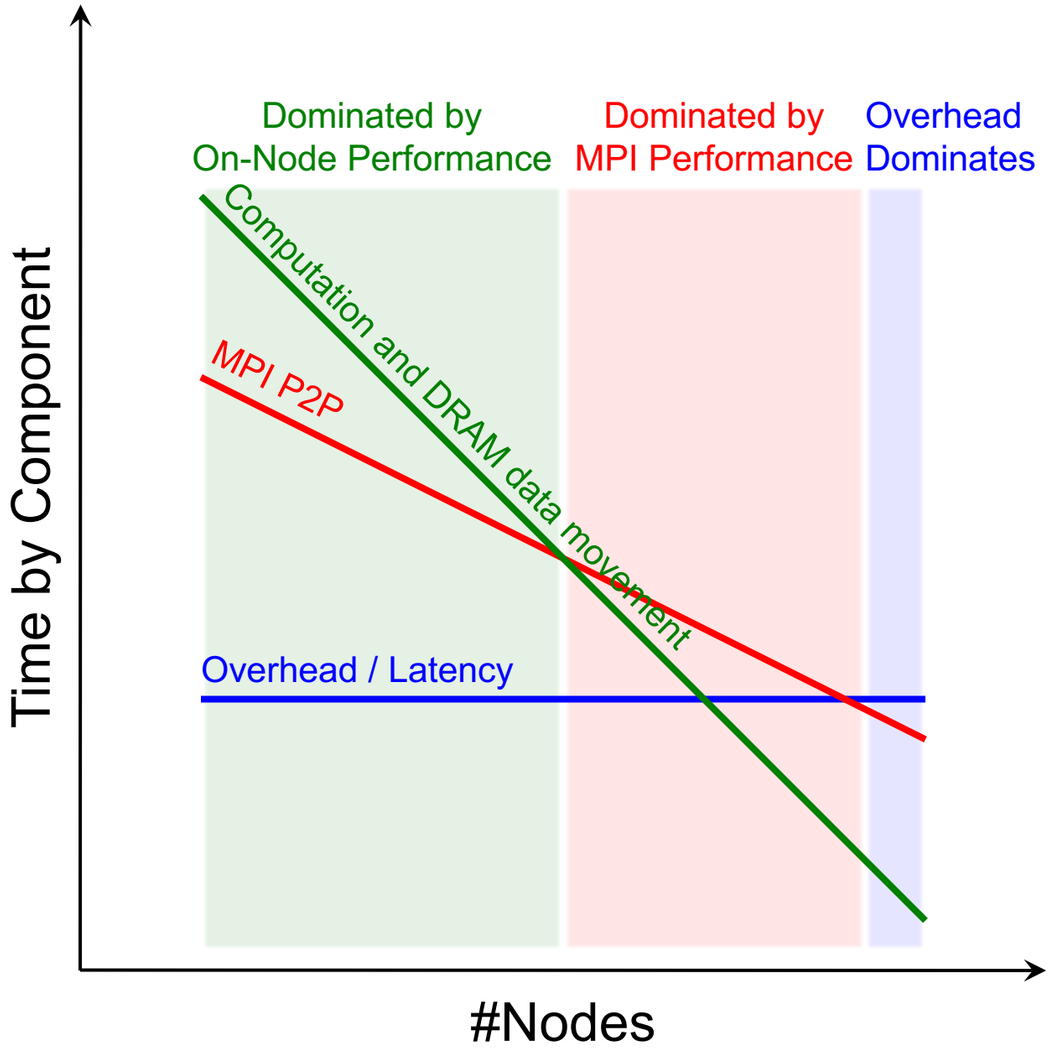
Strong vs. Weak Scaling

- **Strong Scaling:**
  - Global problem sized remains fixed
  - Scale the number of nodes (**per-node problem size decreases with #nodes**)
  - network/threading **overheads/latencies quickly dominate run time**
- **Weak Scaling:**
  - Global problem size grows proportional to the number of nodes
  - Scale the number of nodes (**per-node problem size remains fixed**)
  - Generally keeps network/ threading overheads/latencies in check
  - Superlinear algorithms become extremely time consuming (Dense LU takes a day)
  - Not amenable to all computational domains
  - Corollary: ***scale per-node problem size with per-node throughput***

    (e.g. when moving from a 2P Xeon to a 6xGPU node)

Strong Scaling Example

- e.g. PDE on a structured grid
- Domain decomposition produces an initially favorable surface:volume ratio (MPI:local)
- Strong scaling quickly reduces the volume.
- Surface (e.g. MPI) quickly dominates.
- If scaling continues, overheads can dominate



Performance Models

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*
}
```

DAX
N

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

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

What are the scaling constants?

FFTs: O(NlogN) in the number of

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

MG: O(N) in the number of ele

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

Why did we depart from ideal scaling?

Data Movement Complexity

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

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

Which is more expensive…

Performing Flop's, or

Moving words from memory

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

Machine Balance and Arithmetic Intensity

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

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

- …and arithmetic intensity as the ratio of…

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

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)
FFTs	O(N		O(logN)
CG	O(N		
MG			O(1)
N-body			O(N)

Kernels with AI less than machine balance are ultimately bandwidth-limited

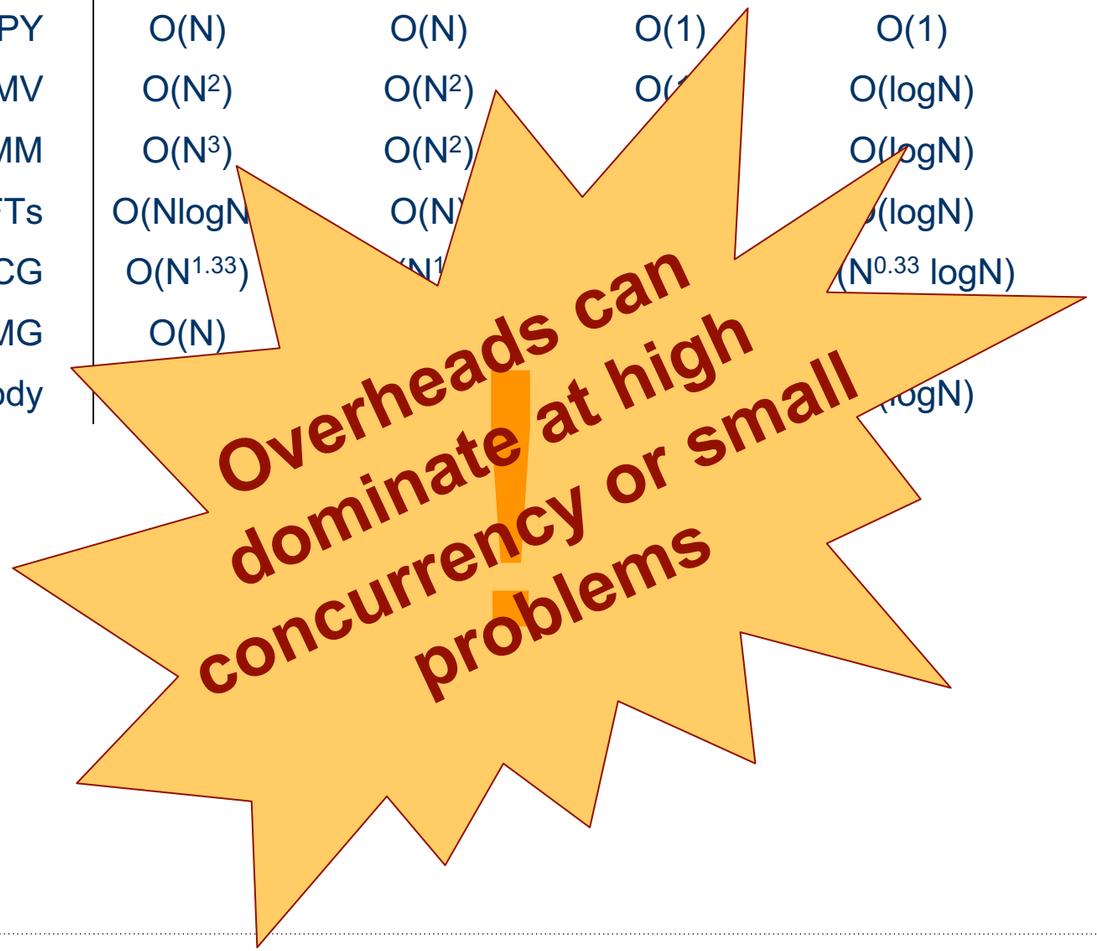
Distributed Memory Performance Modeling

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

Computational Depth

- Imagine a world of infinite parallelism & bandwidth, but finite latencies
- We can classify algorithms by **depth** (max depth of the algorithm's dependency chain)
- For iterative algorithms, this is product of iterations and depth per iteration

Operation	Flop's	Data	AI (ideal)	Depth
DAXPY	O(N)	O(N)	O(1)	O(1)
DGEMV	$O(N^2)$	$O(N^2)$	O(	O(logN)
DGEMM	$O(N^3)$	$O(N^2)$		O(logN)
FFTs	O(NlogN	O(N		(logN)
CG	$O(N^{1.33})$	(N1		$(N^{0.33}$ logN)
MG	O(N)			
N-body				logN)



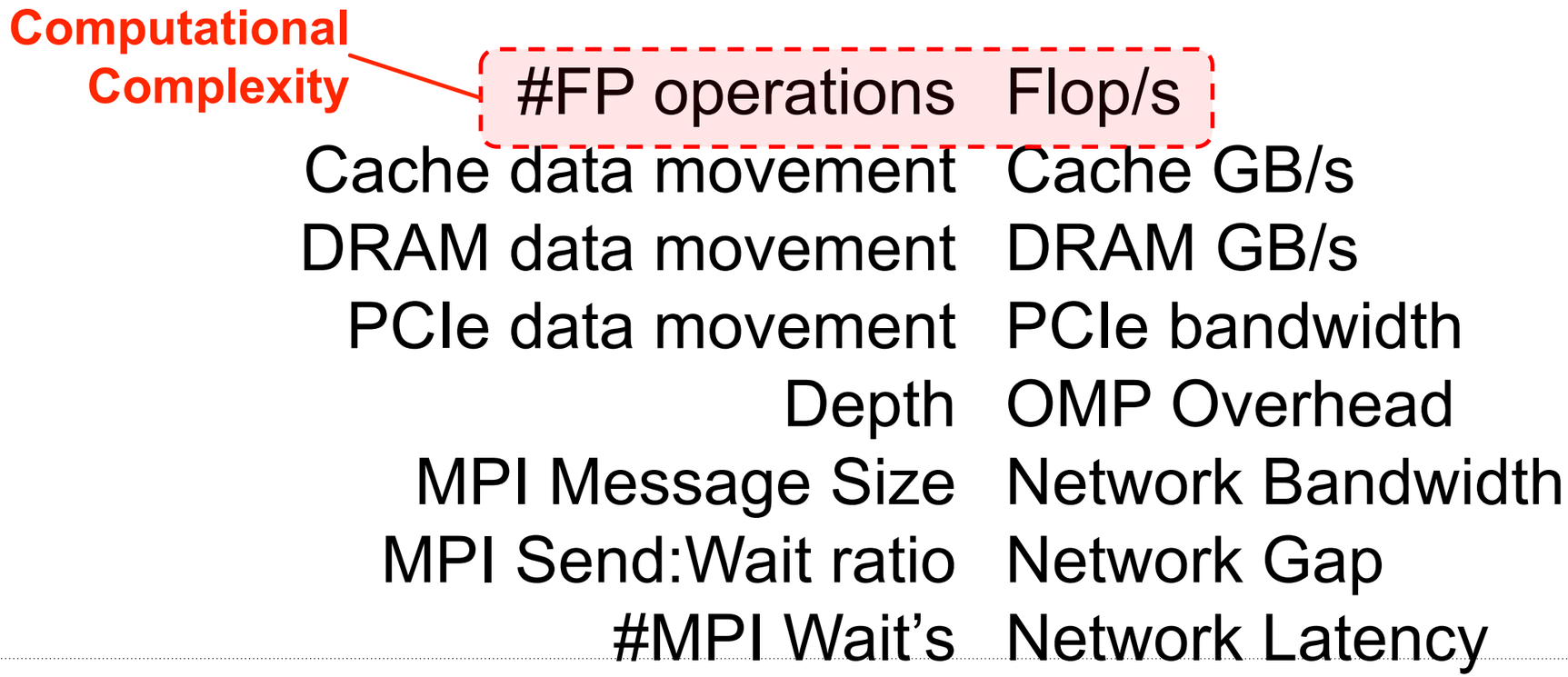
Performance Models

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

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

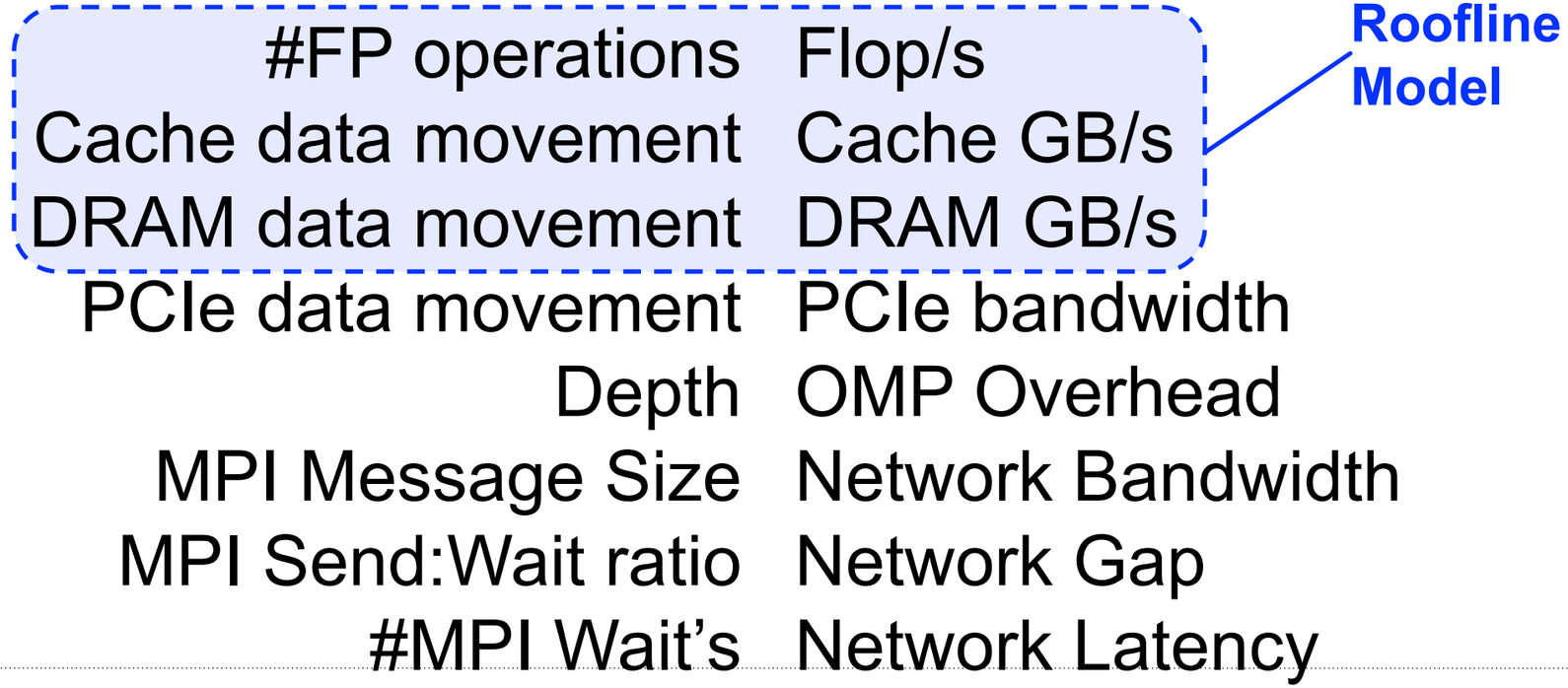
Performance Models

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



Performance Models

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



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

Performance Models

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

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

LogP

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

Performance Models

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

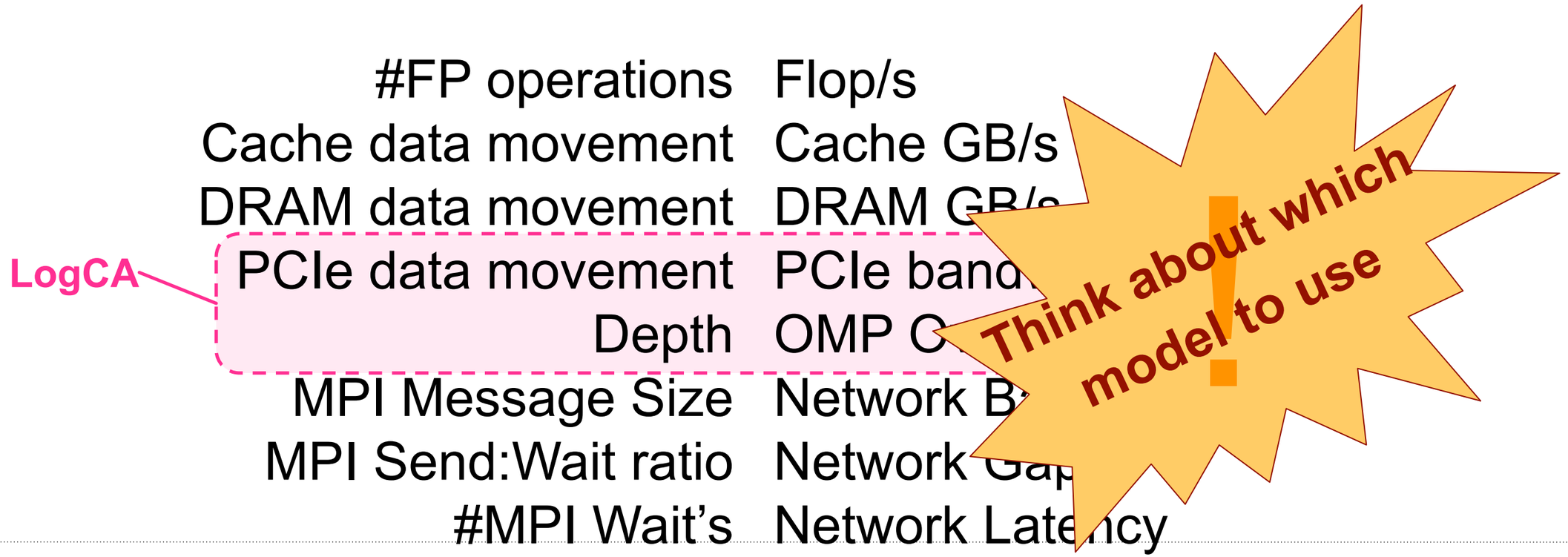
#FP operations	Flop/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe bandwidth
Depth	OMP Overhead
MPI Message Size	Network Bandwidth
MPI Send:Wait ratio	Network Gap
#MPI Wait's	Network Latency

LogGP

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

Performance Models

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



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

Introduction to the Roofline Model

Performance Models / Simulators

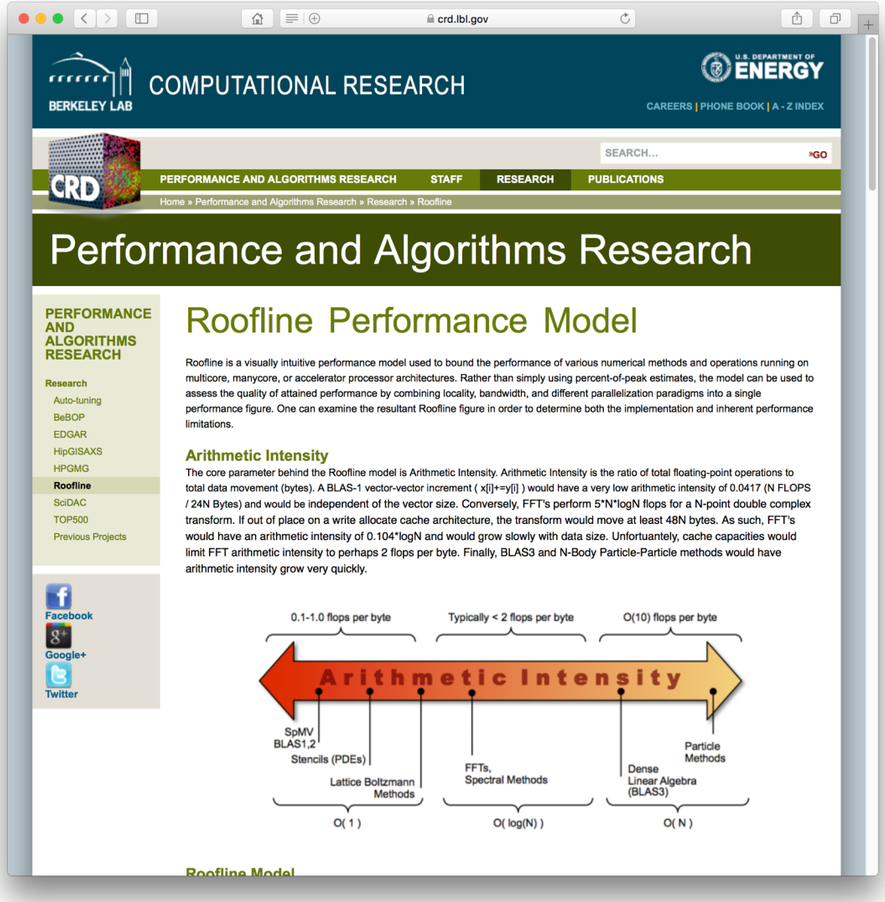
- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)

- 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 TPUs[1], etc…)

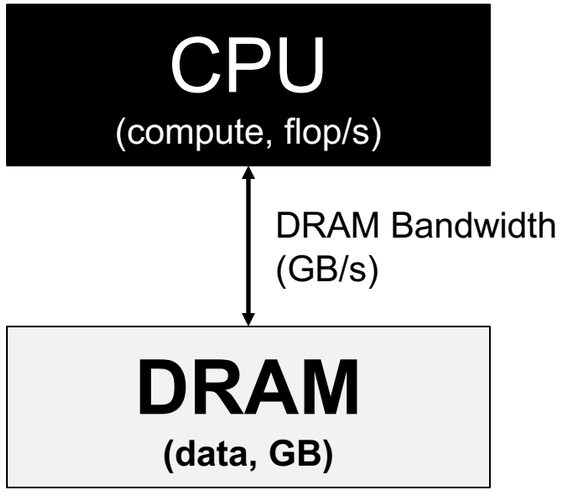


https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

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

(DRAM) Roofline

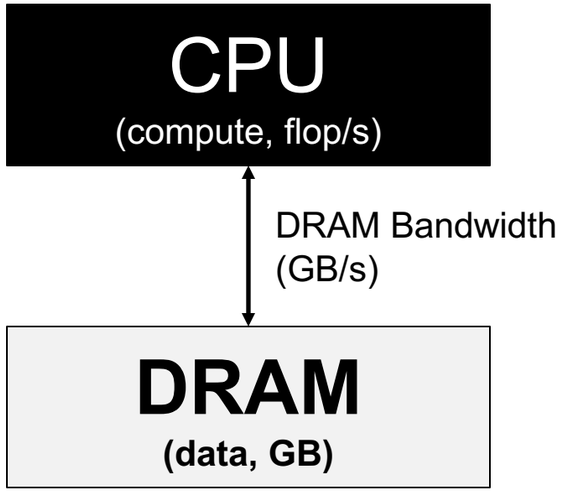
- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



Time = max { #FP ops / Peak GFlop/s ; #Bytes / Peak GB/s }

(DRAM) Roofline

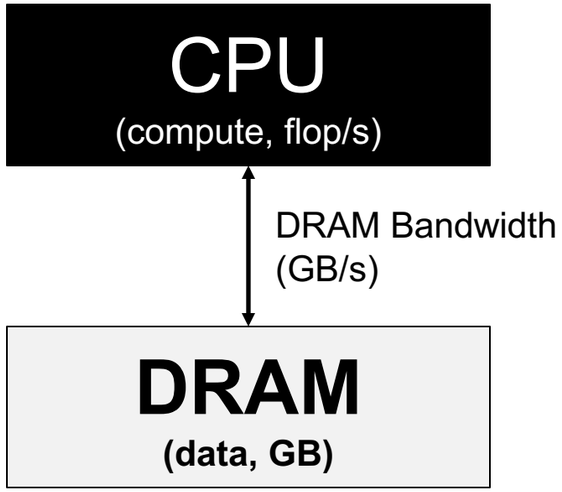
- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

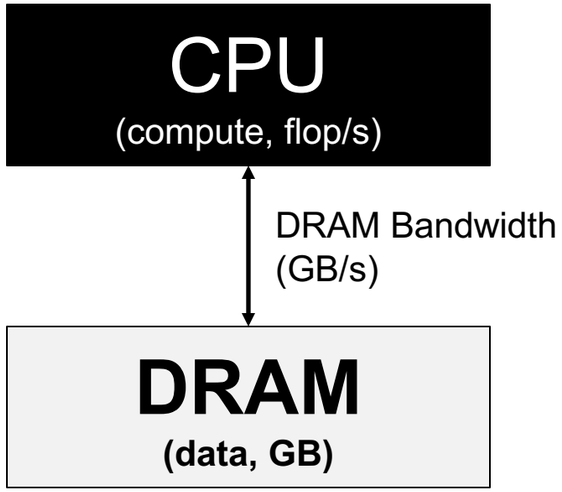
- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

(DRAM) Roofline

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

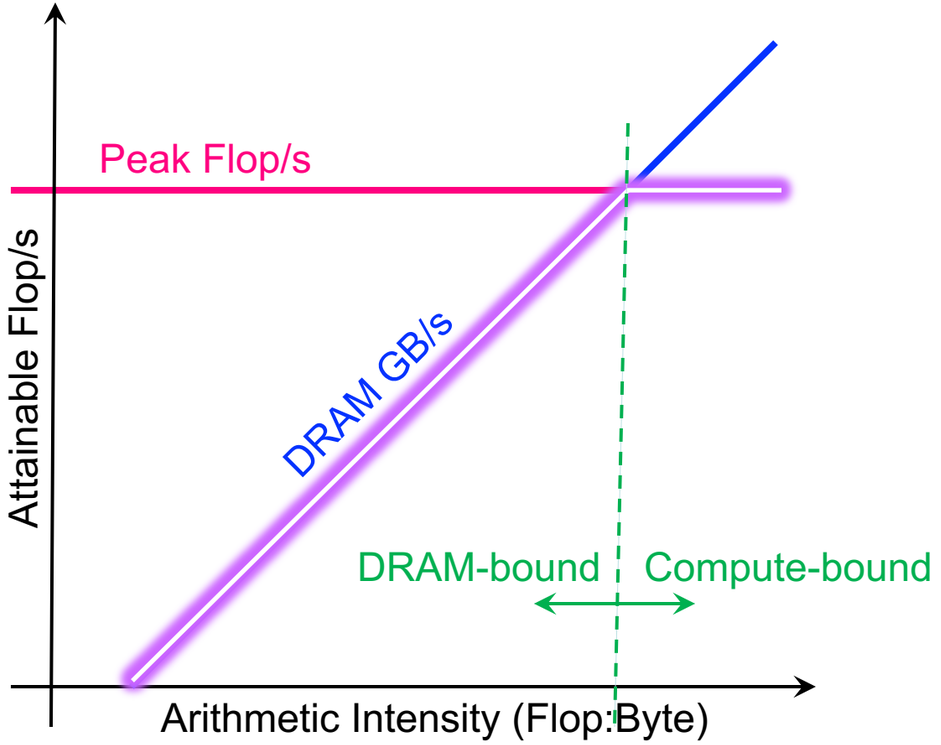


GFlop/s = min { Peak GFlop/s ; AI * Peak GB/s }

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

(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…)



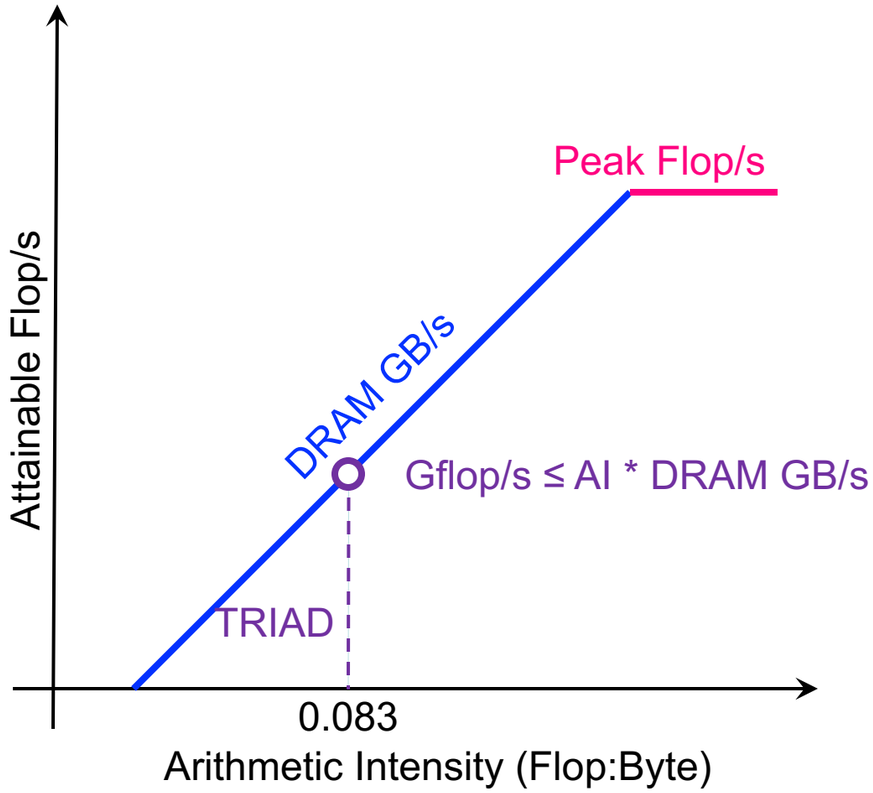
Roofline Example #1

- 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**

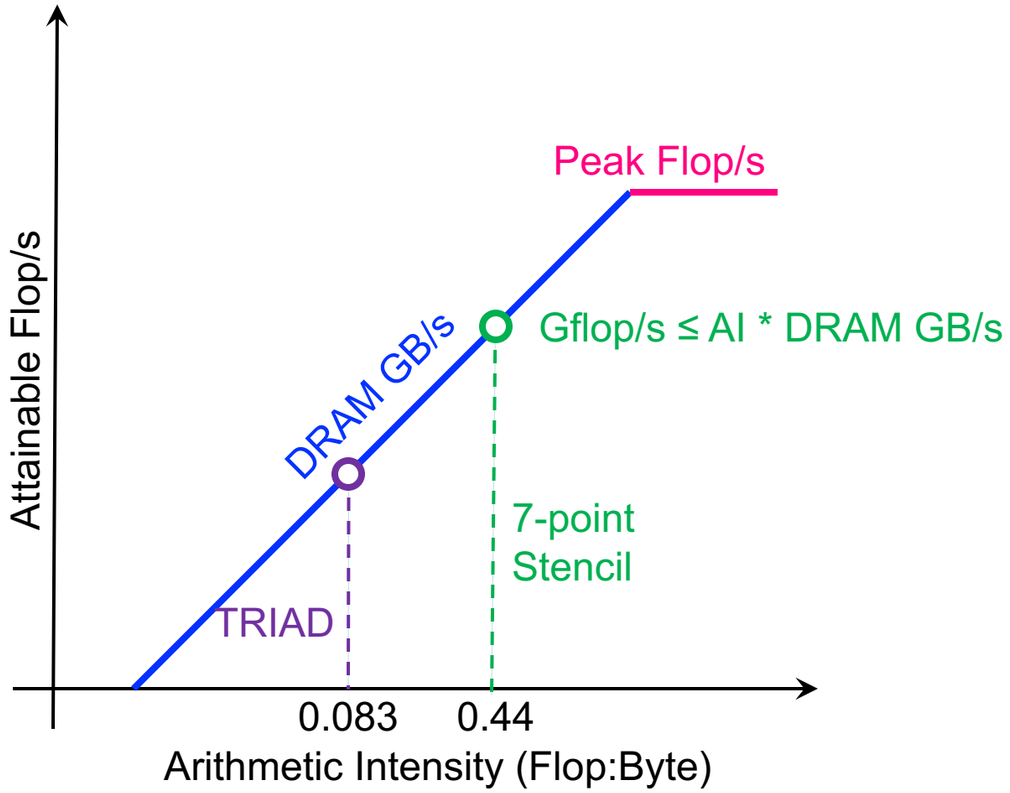


Roofline Example #2

- 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];
}}}
```

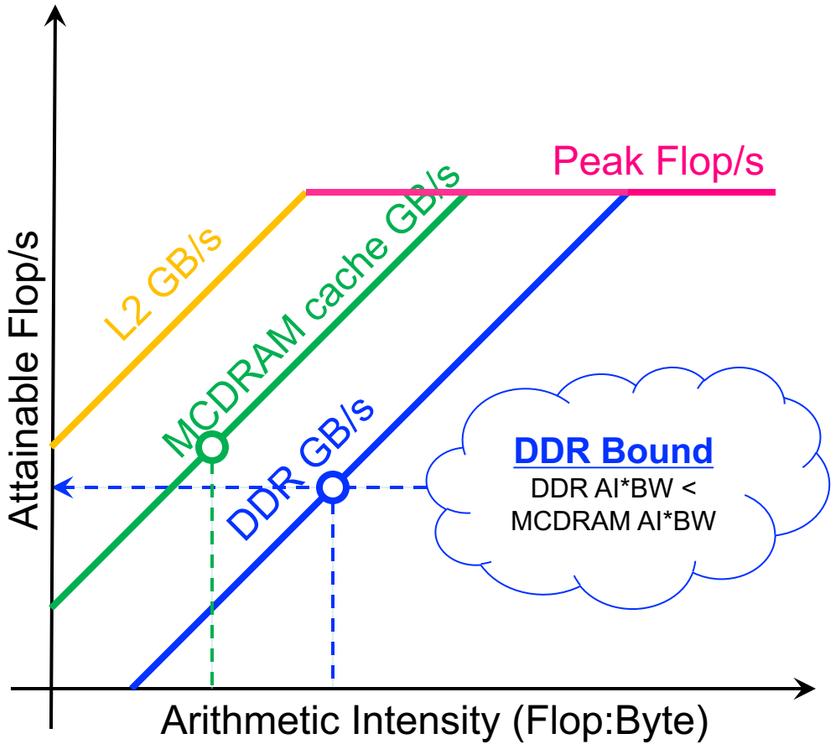


Hierarchical Roofline

- 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

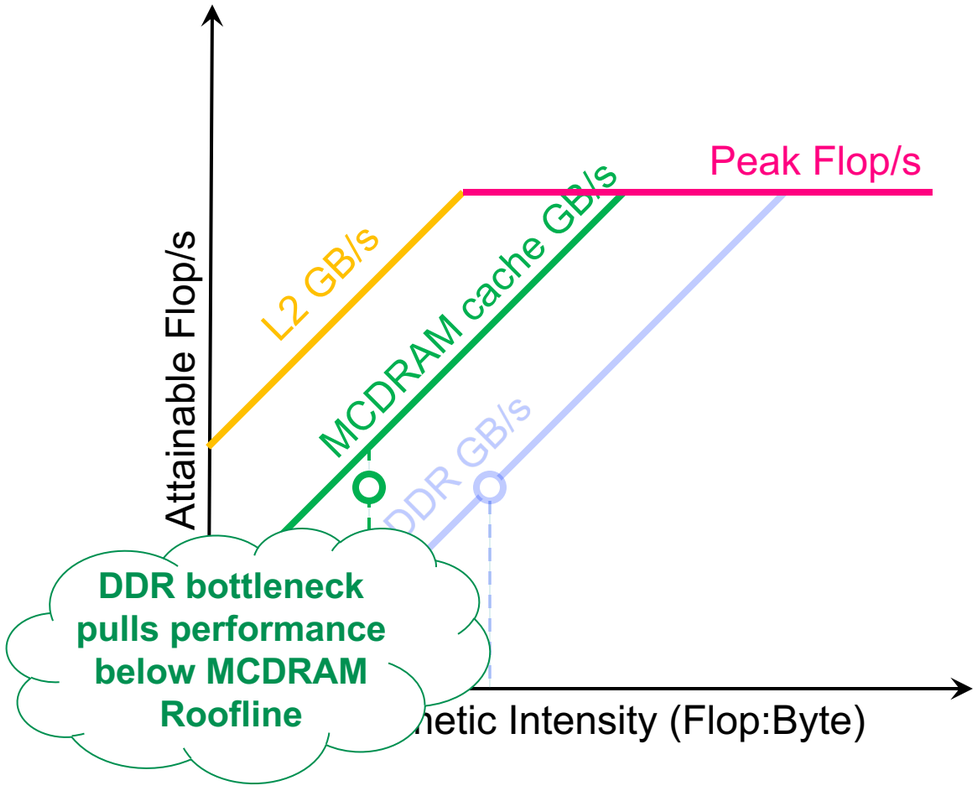
Hierarchical Roofline

- 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**



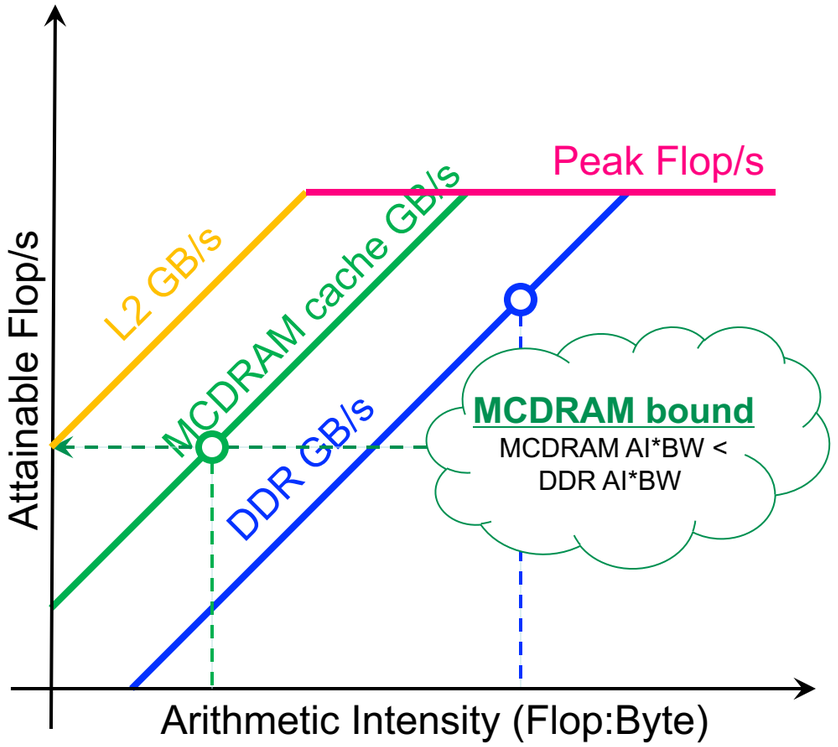
Hierarchical Roofline

- 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**



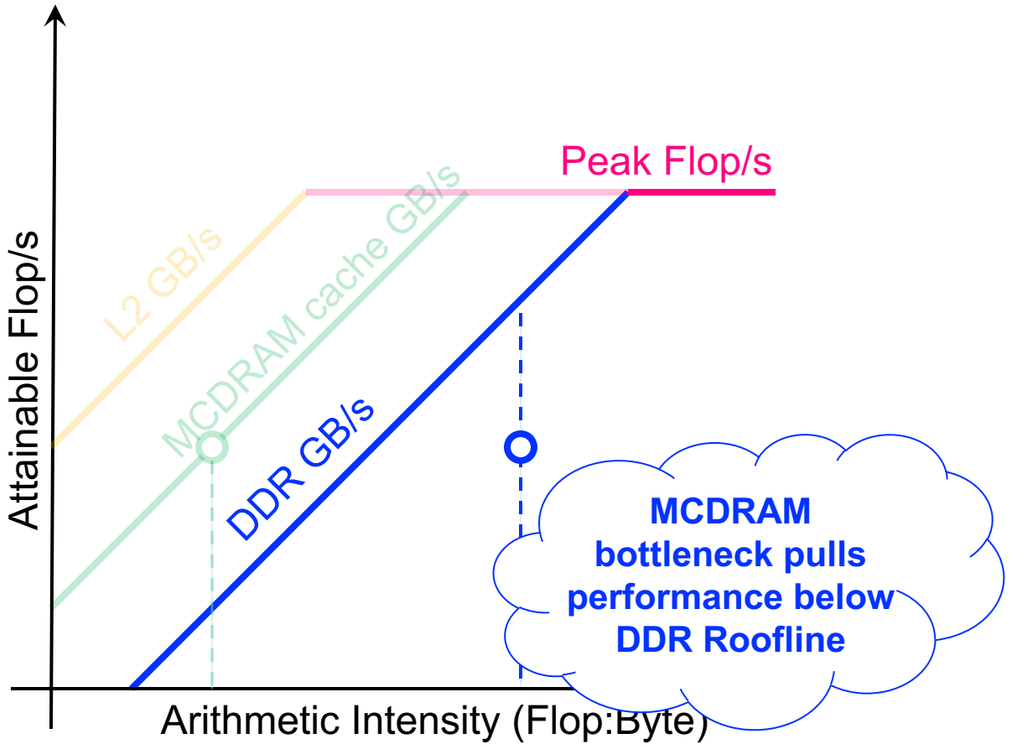
Hierarchical Roofline

- 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**



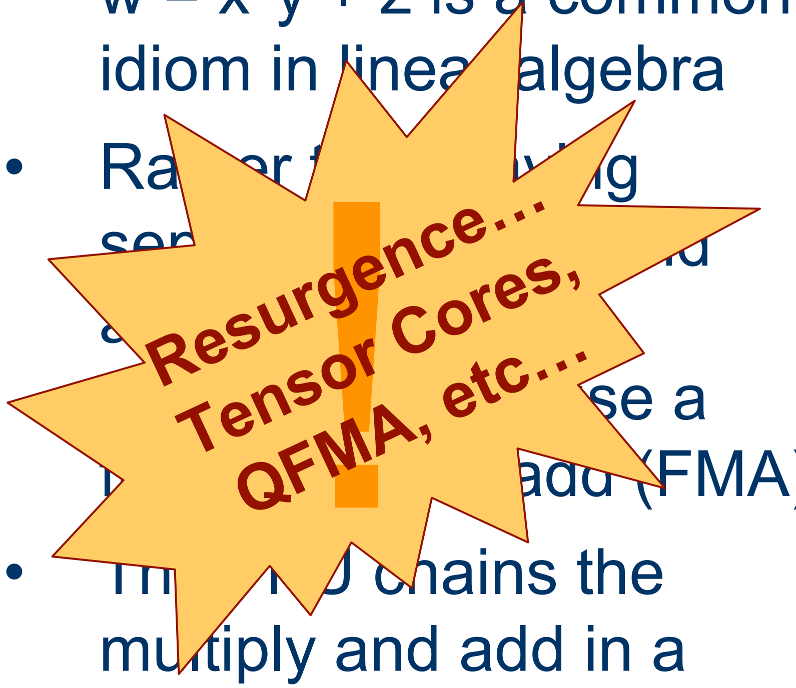
Roofline Model: In-Core Effects

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
- Ra… er … ving se… d a… se a … add (FMA)
- … chains the multiply and add in a single pipeline so that it can complete FMA/cycle



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…

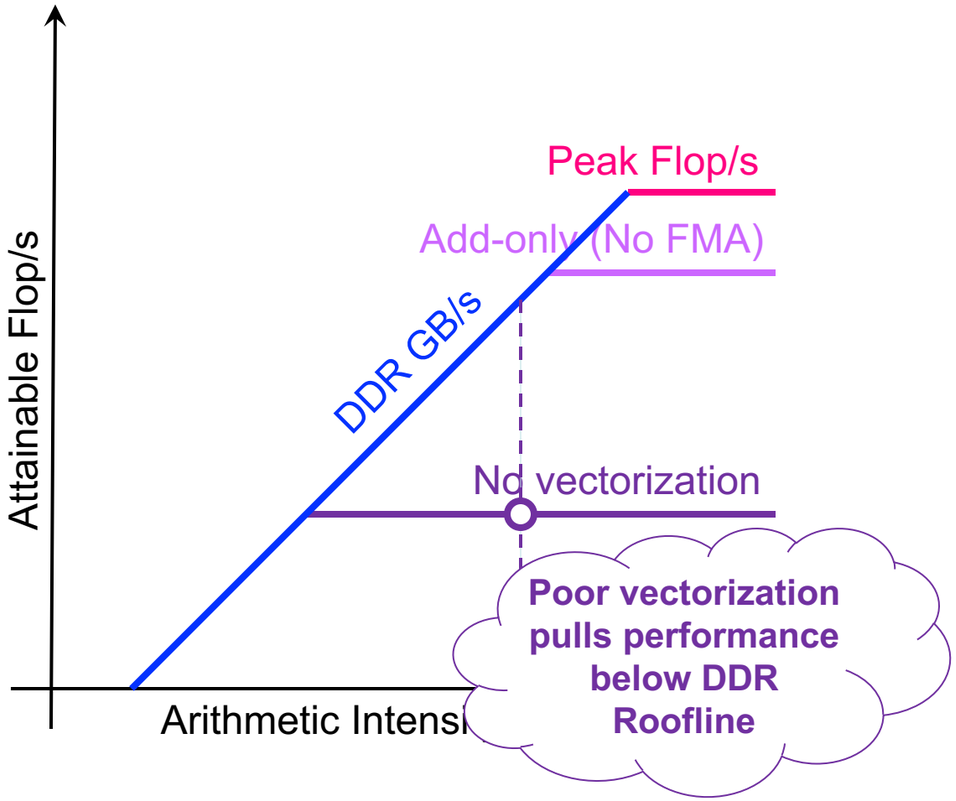
  x [0:7] *y [0:7] + z [0:7]
- Vector FPUs complete 8 vector operations/cycle

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

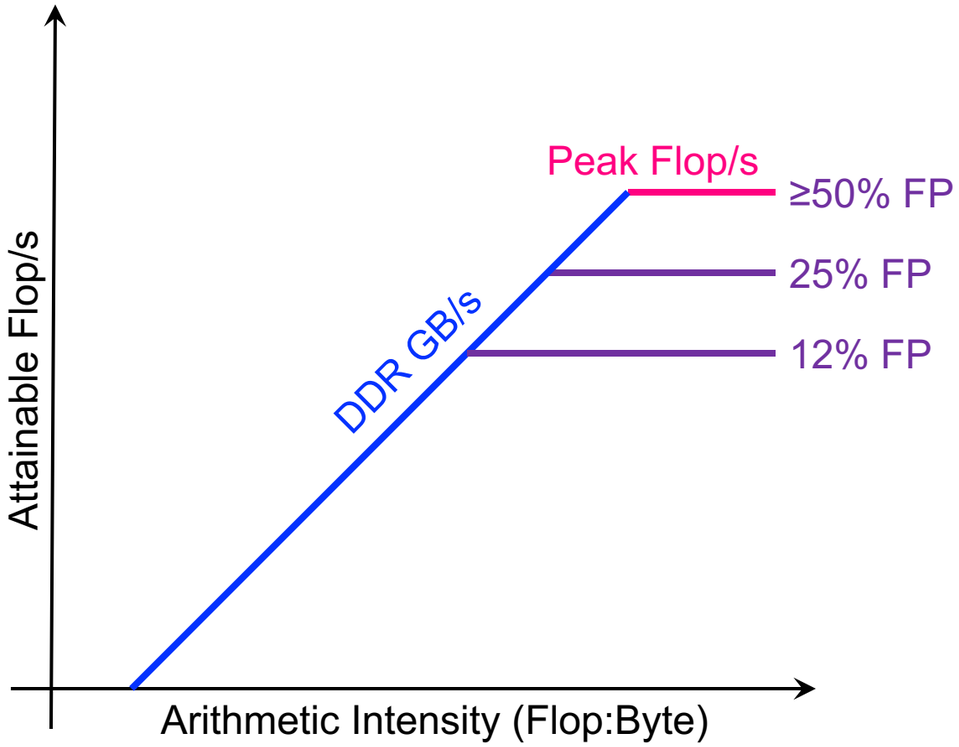
Data, Instruction, Thread-Level Parallelism…

- 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.



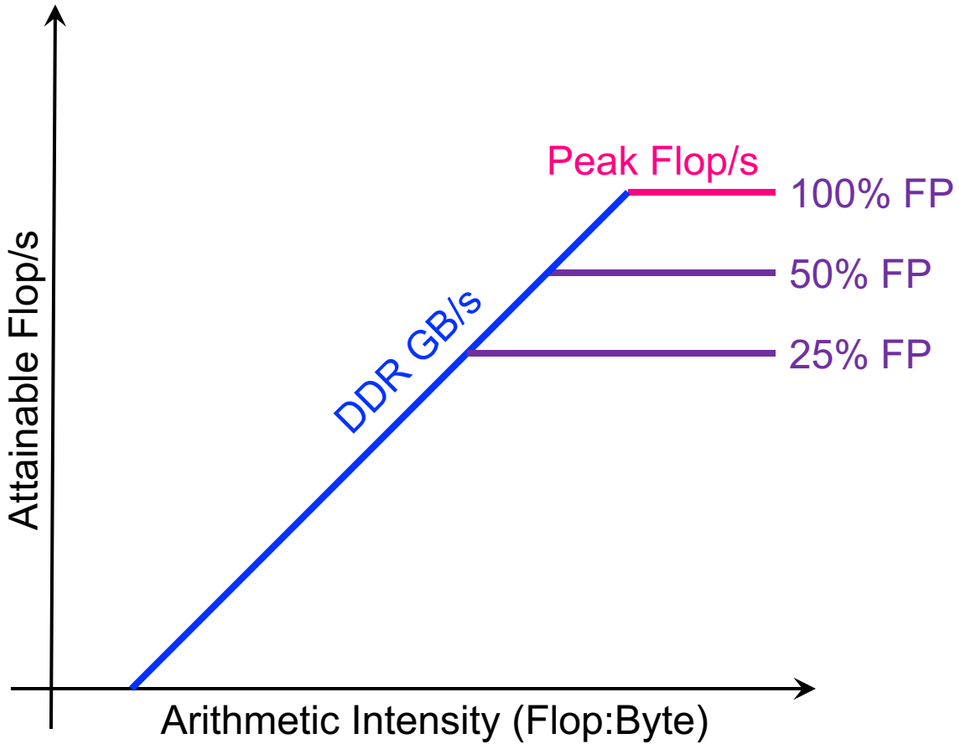
Superscalar vs. instruction mix

- 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



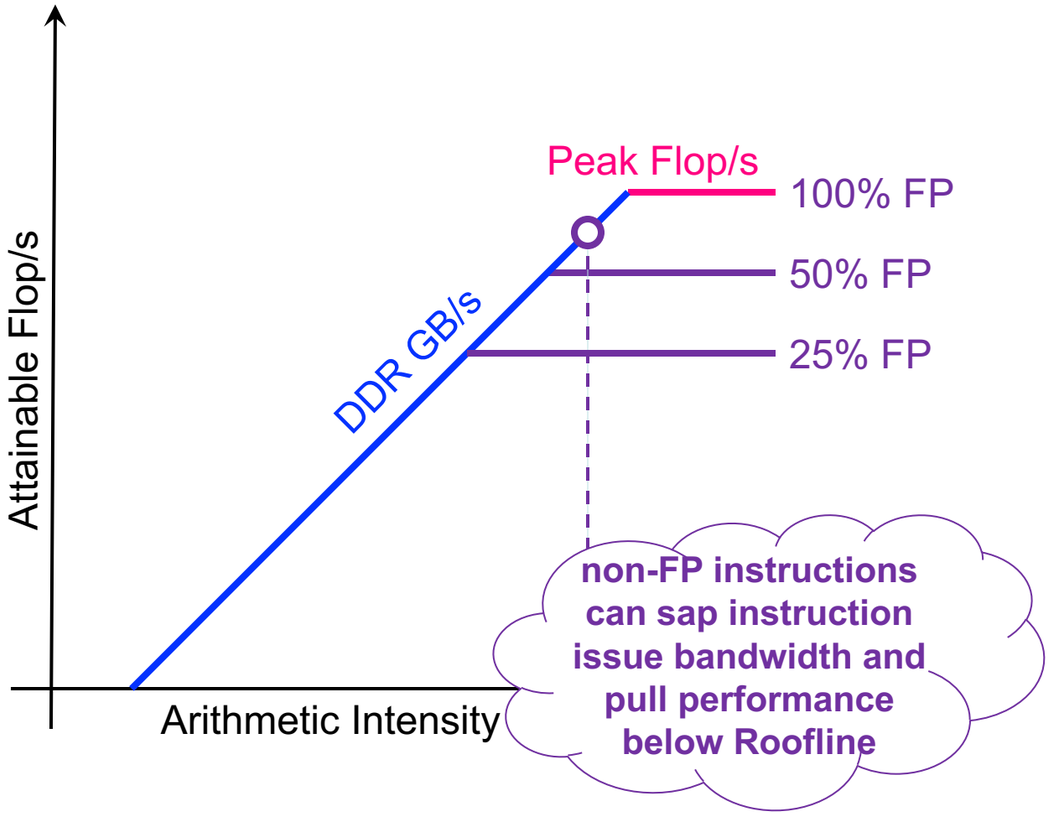
Superscalar vs. instruction mix

- 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

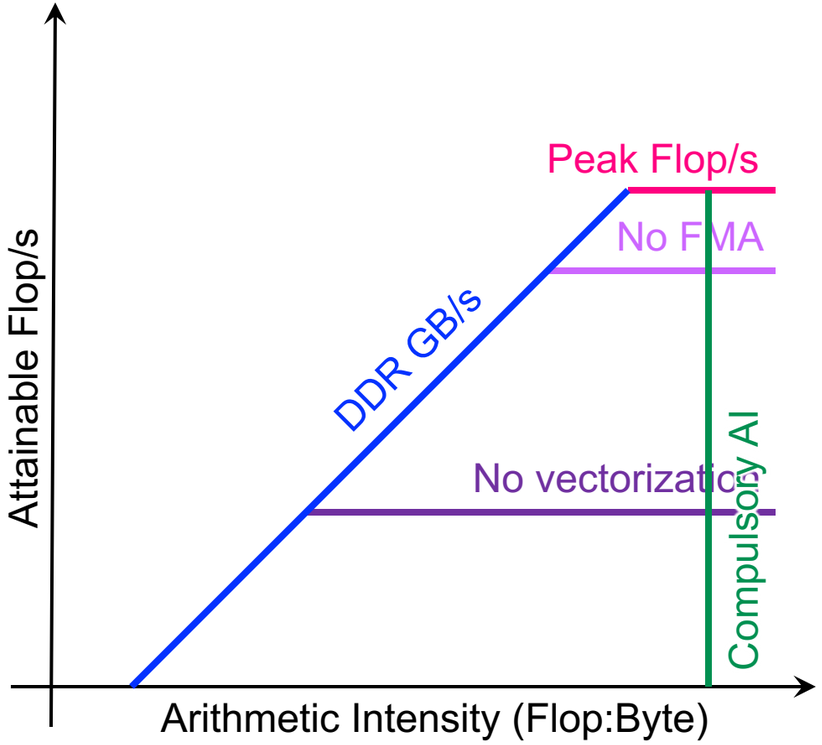
- Define in-core ceilings based on instruction mix…
- e.g. Haswell
  - 4-issue superscalar
  - Only 2 FP data paths
  - Requires 50% of the instructions to be FP to get peak performance
- e.g. KNL
  - 2-issue superscalar
  - 2 FP data paths
  - Requires 100% of the instructions to be FP to get peak performance



Roofline Model: Cache Effects

Locality Walls

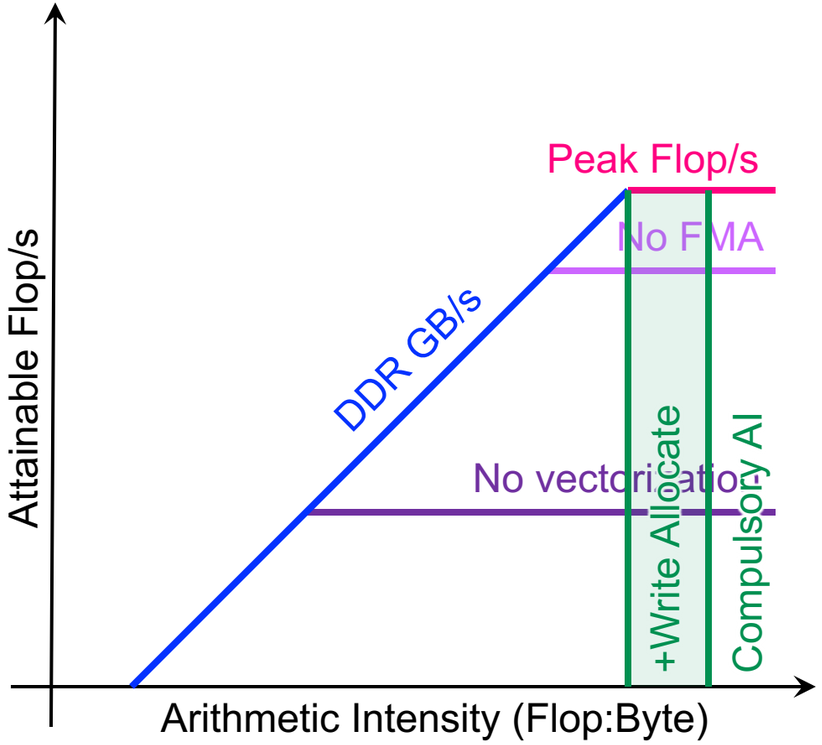
- Naively, we can bound AI using only compulsory cache misses



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

Locality Walls

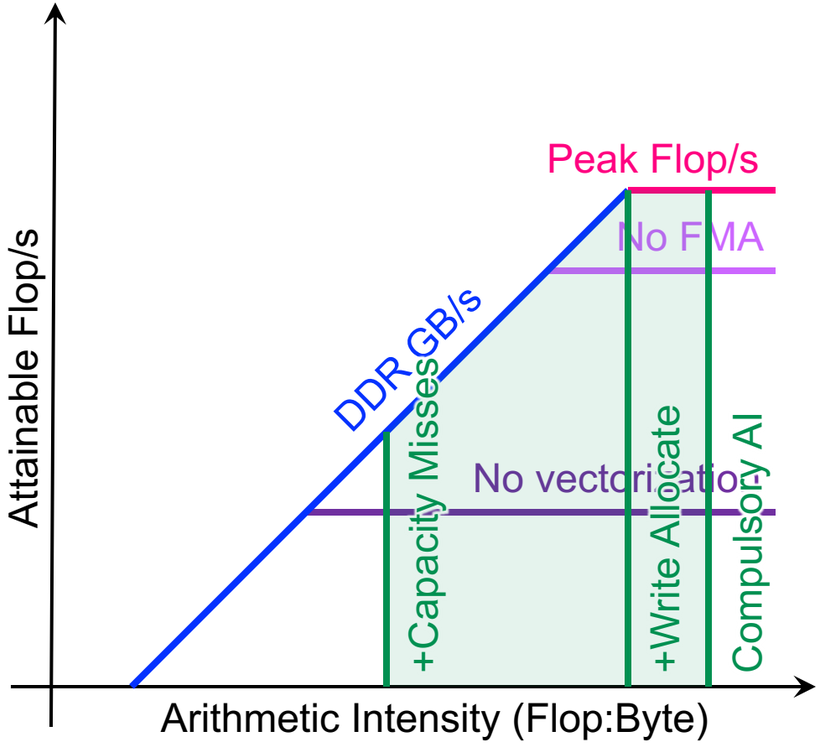
- Naively, we can bound AI using only compulsory cache misses
- However, write allocate caches can lower AI



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

Locality Walls

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

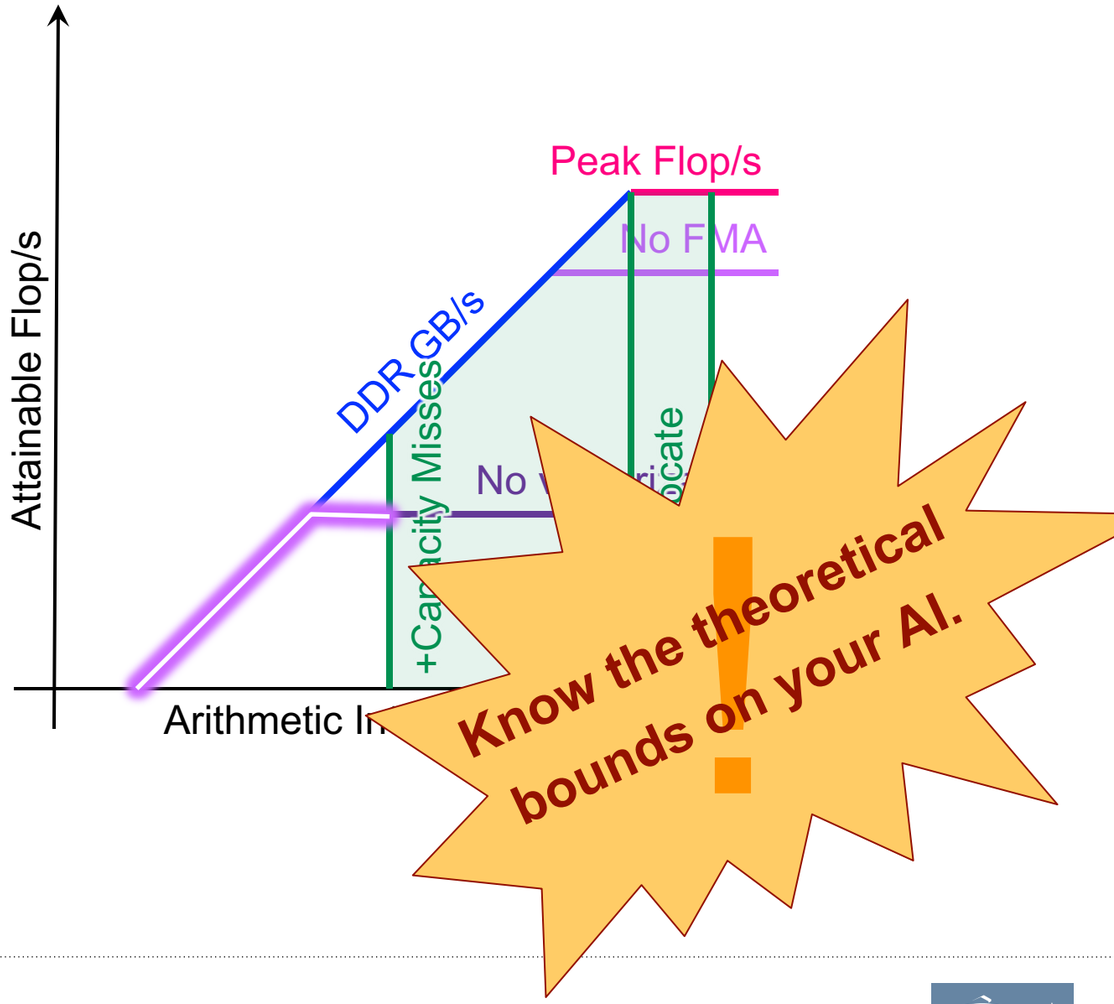


$$AI = \frac{\#Flop's}{Compulsory\ Misses + Write\ Allocates + Capacity\ Misses}$$

Locality Walls

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

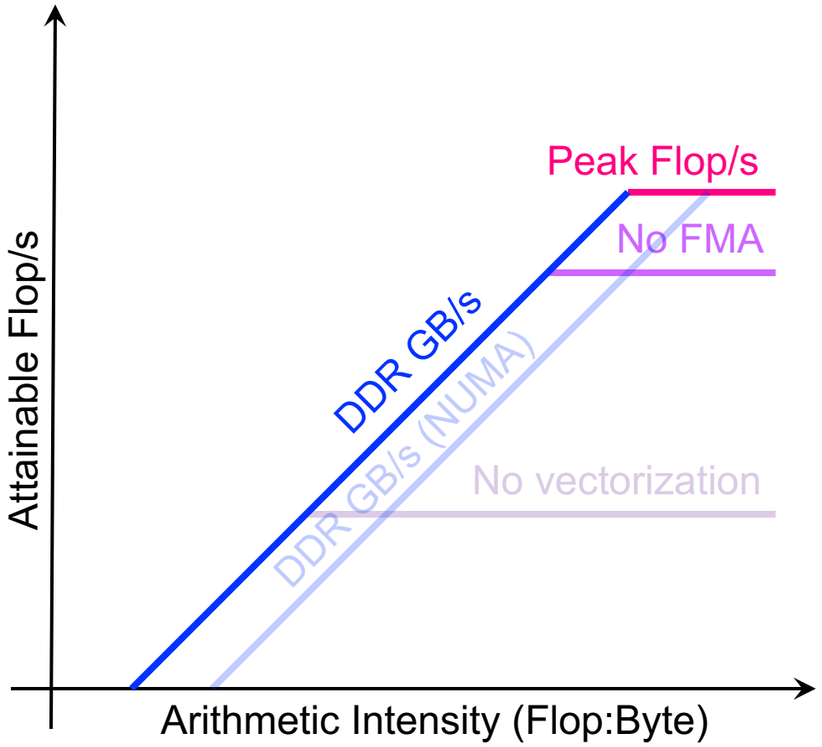
$$AI = \frac{\#Flop's}{Compulsory\ Misses + Write\ Allocates + Capacity\ Misses}$$



Roofline Model: General Strategy Guide

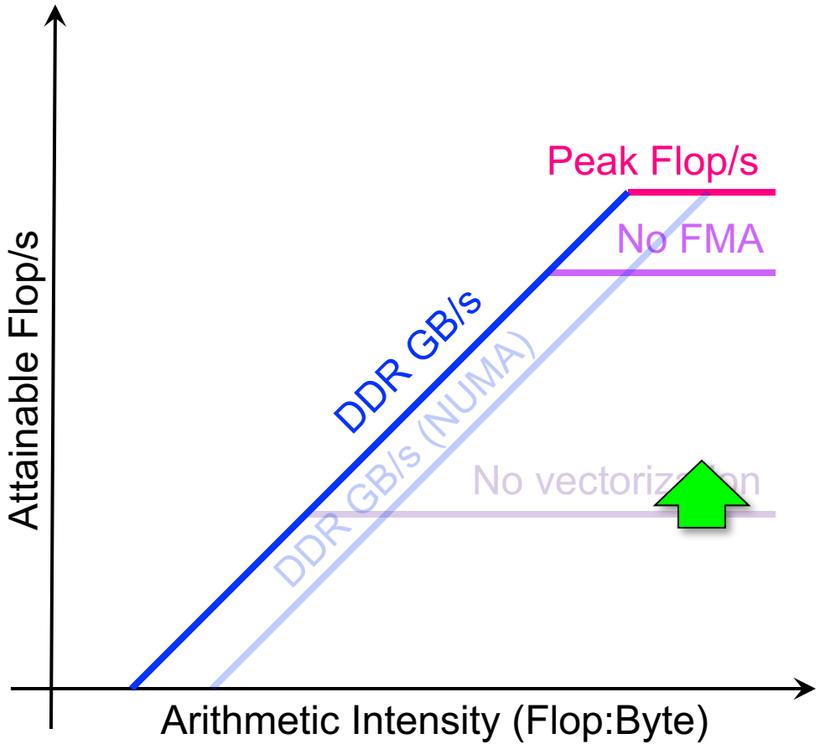
General Strategy Guide

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



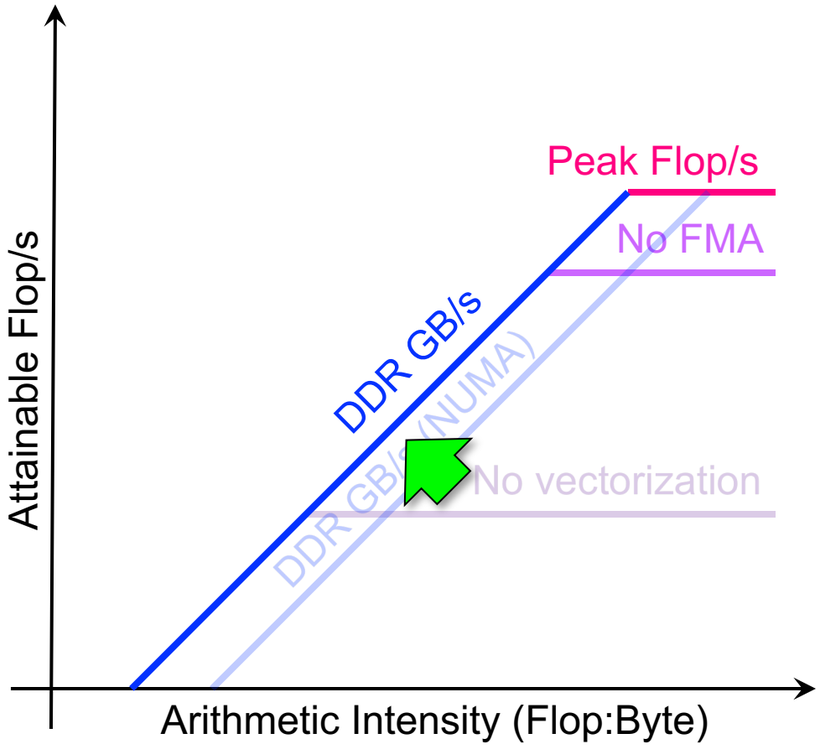
General Strategy Guide

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



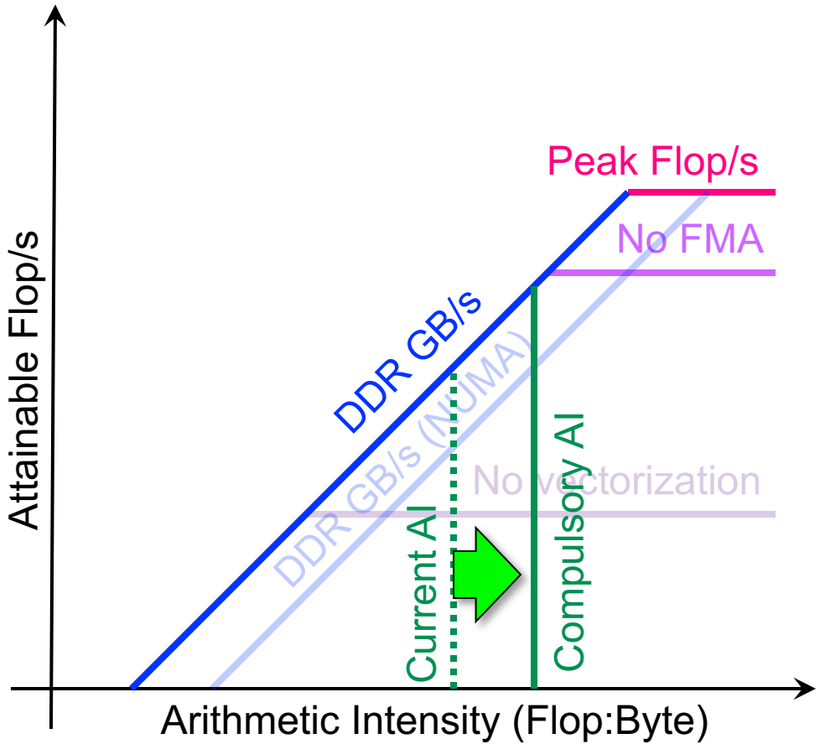
General Strategy Guide

- 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)**



Performance Tools

Overview

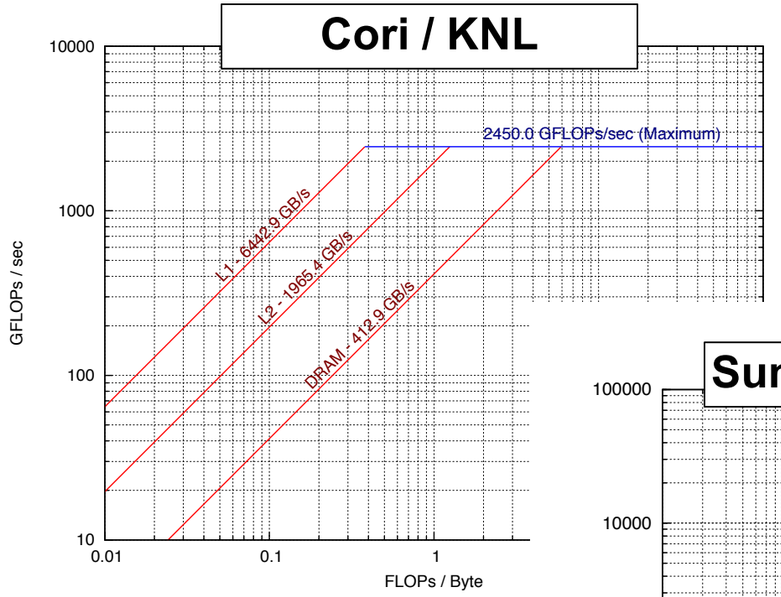
- Machine Characterization
- Application Instrumentation (timing breakdowns)
- Performance Analysis

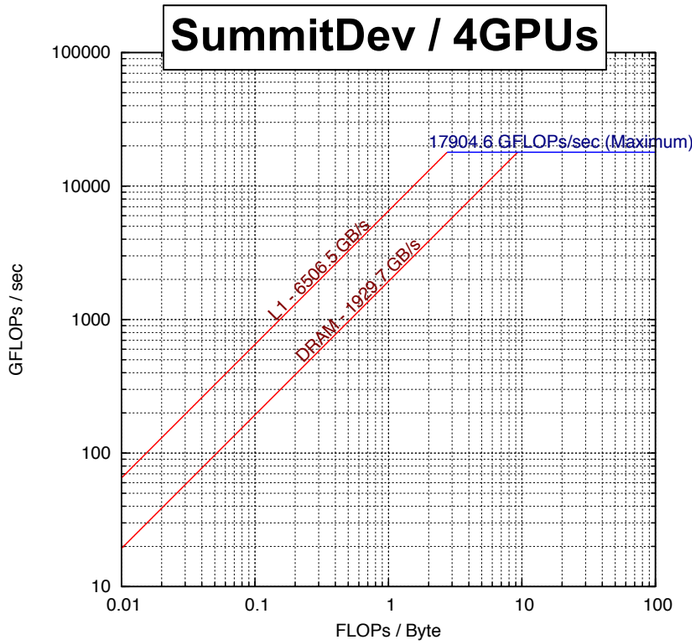
- Timers
- Performance Counters
- Simulators / Code introspection (slow)
- Sampling (data is meaningless if it falls below sampling granularity)

- Timeline
- Time-integrated (seconds)
- Throughput-oriented (Gflop/s or GB/s)

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**





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

Manual Instrumentation

- Application developers know best…
  - What to time
  - What to record (e.g. solver iterations, dimensions, etc…)
  - How asynchrony is exploited
  - How the same function might be called/used many different ways == unique timers
  - Computational load imbalance (i.e. understand why timing shows imbalance)
- **Make timing instrumentation a first-class citizen with developing an application**
- Create timing wrapper that uses either…
  - omp_get_wtime()
  - MPI_Wtime()
  - rdtsc or equivalent

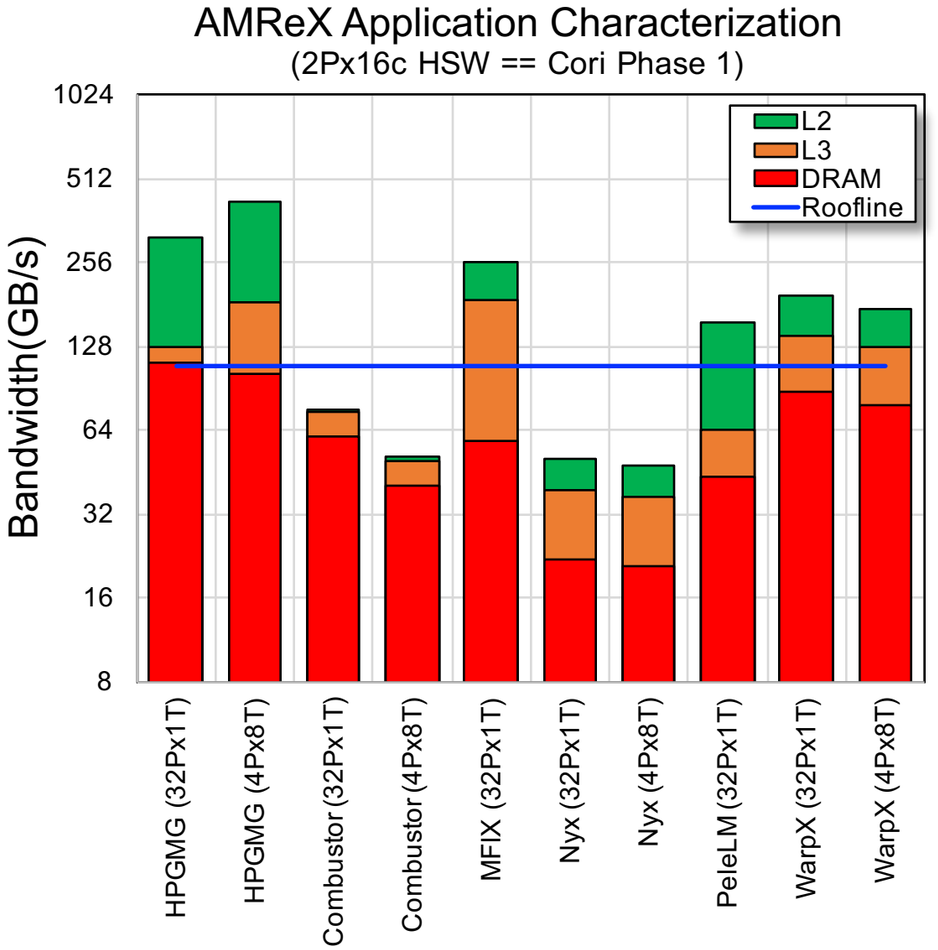
LIKWID

- 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**

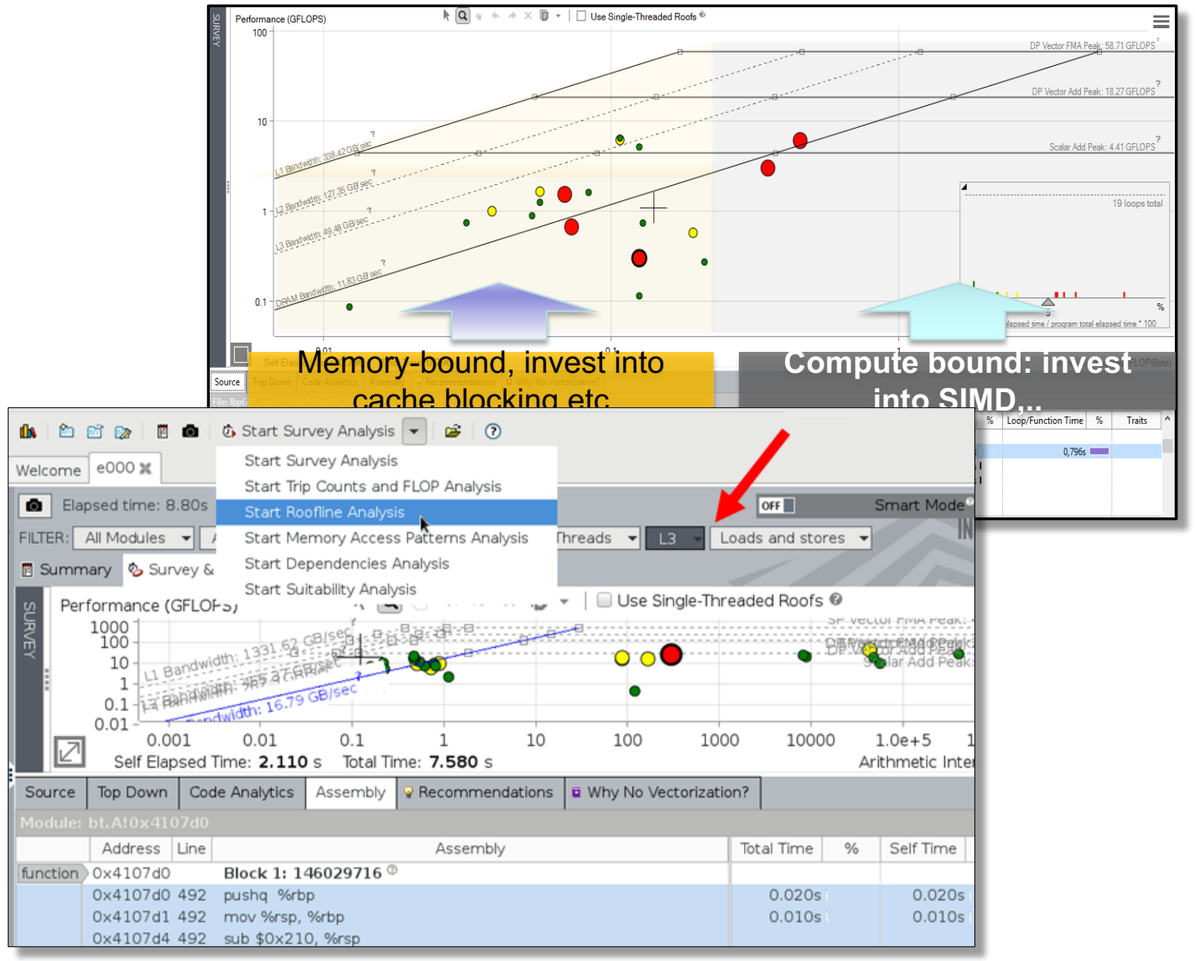
https://github.com/RRZE-HPC/likwid

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



Intel Advisor

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

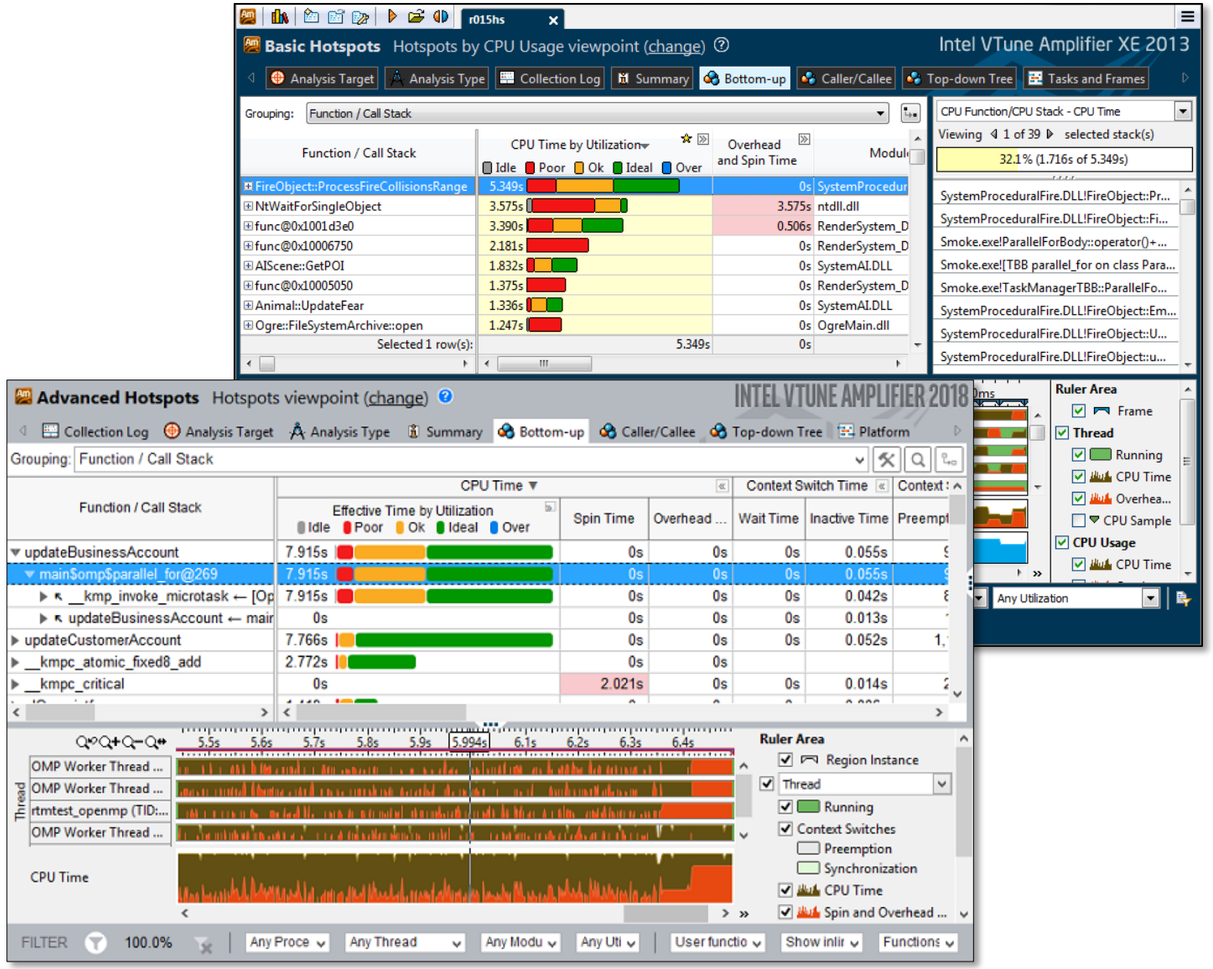


http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017

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

Intel VTune

- ✓ Automatically instruments applications (sampling)
- ✓ Presents time-oriented execution
- ✓ Has access to performance counters (e.g. %bandwidth, front-end bound, etc…)
- ✓ Hotspot, Concurrency, Synchronization, and Memory Analysis Options



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

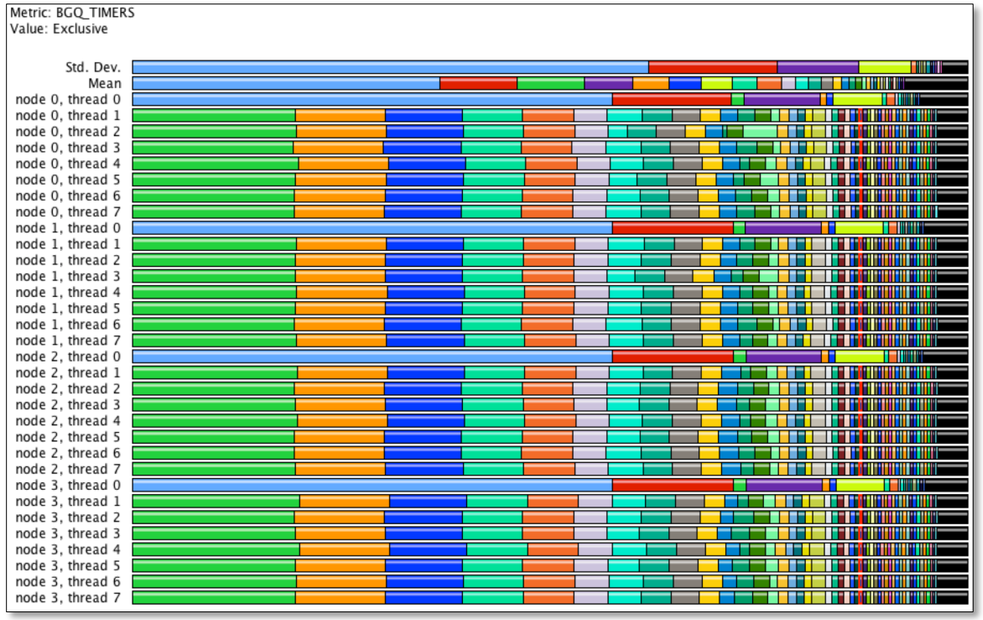
Cray PAT

- ✓ Automatically instruments applications (sampling)
- ✓ Tracing support for user-specified functions
- ✓ Inherent distributed memory (MPI) support
- ✓ Presents time-integrated results
- ✓ Has access to performance counters (e.g. %bandwidth, flop/s, etc…)
- ✓ Load imbalance, MPI, OpenMP, etc…
- ✓ Text output (but Cray Reveal can visualize output)

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

TAU

- ✓ Automatically instruments applications
- ✓ Sampling and Tracing support
- ✓ Presents time-oriented or time-integrated results
- ✓ Visualization of MPI communication & load imbalance
- ✓ Integrates with MPI (inherent distributed memory support)
- ✓ Can leverage HW performance counters



https://www.cs.uoregon.edu/research/tau/home.php

Questions?

Backup

Hierarchical Roofline vs. Cache-Aware Roofline

…understanding different Roofline formulations in Advisor

There are two Major Roofline Formulations:

- Hierarchical Roofline (original Roofline w/ DRAM, L3, L2, …)…
  - **Williams, et al, "Roofline: An Insightful Visual Performance Model for Multicore Architectures", CACM, 2009**
  - **Chapter 4 of "Auto-tuning Performance on Multicore Computers", 2008**
  - Defines multiple bandwidth ceilings and multiple AI's per kernel
  - Performance bound is the minimum of flops and the memory intercepts (superposition of original, single-metric Rooflines)
- Cache-Aware Roofline
  - **Ilic et al, "Cache-aware Roofline model: Upgrading the loft", IEEE Computer Architecture Letters, 2014**
  - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
  - As one looses cache locality (capacity, conflict, …) performance falls from one BW ceiling to a lower one at constant AI
- Why Does this matter?
  - Some tools use the Hierarchical Roofline, some use cache-aware **== Users need to understand the differences**
  - Cache-Aware Roofline model was integrated into production Intel Advisor
  - Evaluation version of Hierarchical Roofline[1] (cache simulator) has also been integrated into Intel Advisor

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

Hierarchical Roofline

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

Cache-Aware Roofline

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

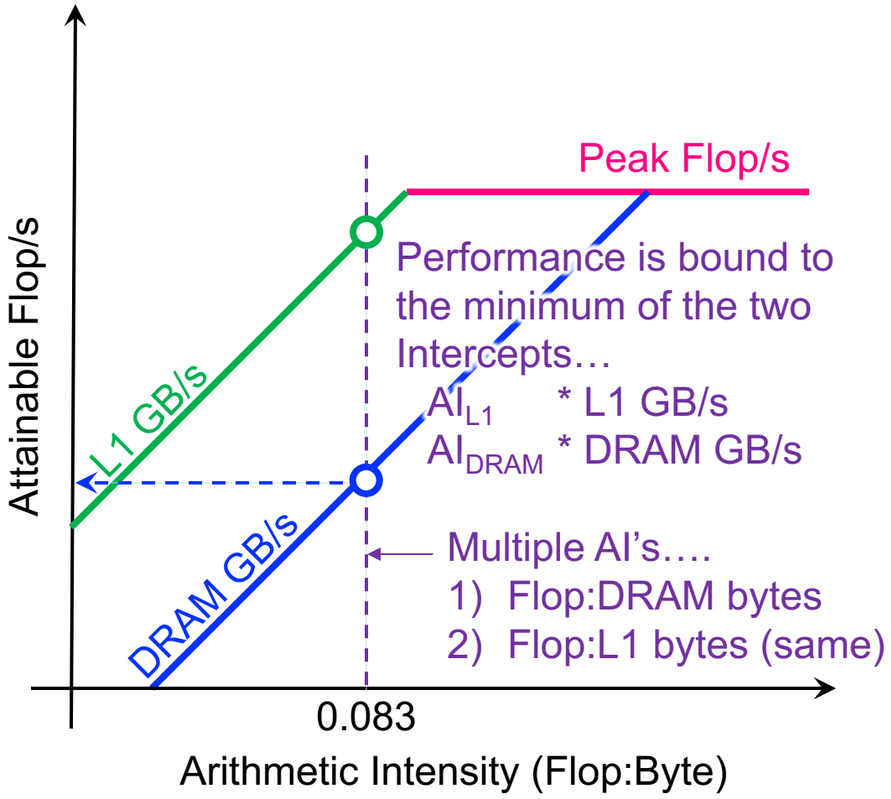
Example: STREAM

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

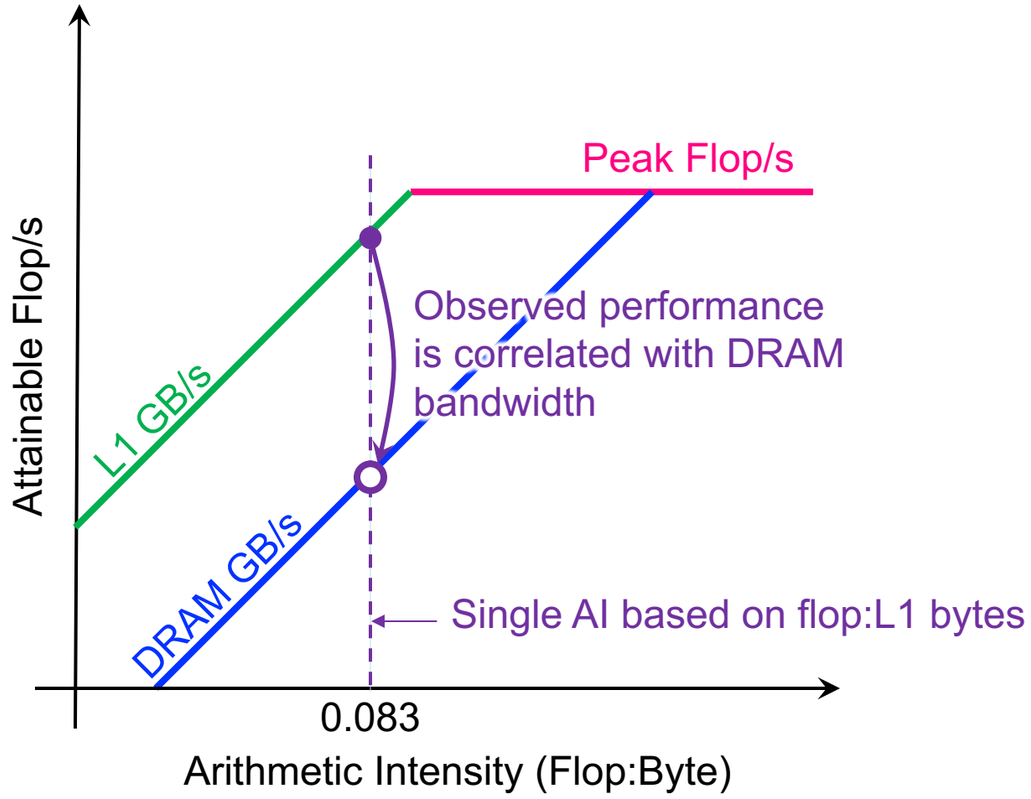
```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

Hierarchical Roofline



Cache-Aware Roofline



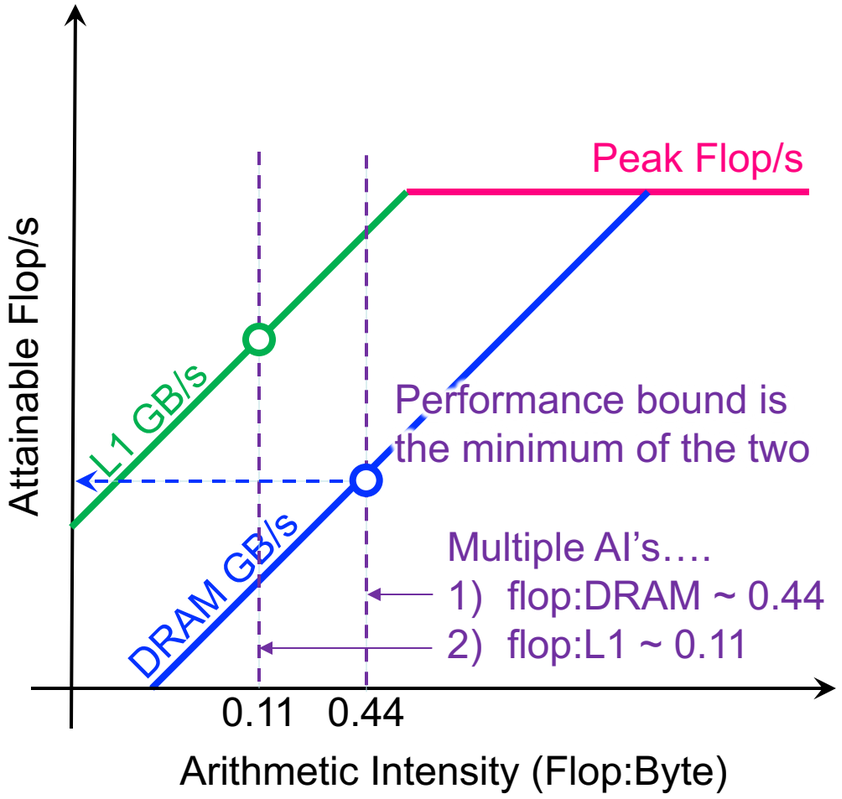
Example: 7-point Stencil (Small Problem)

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

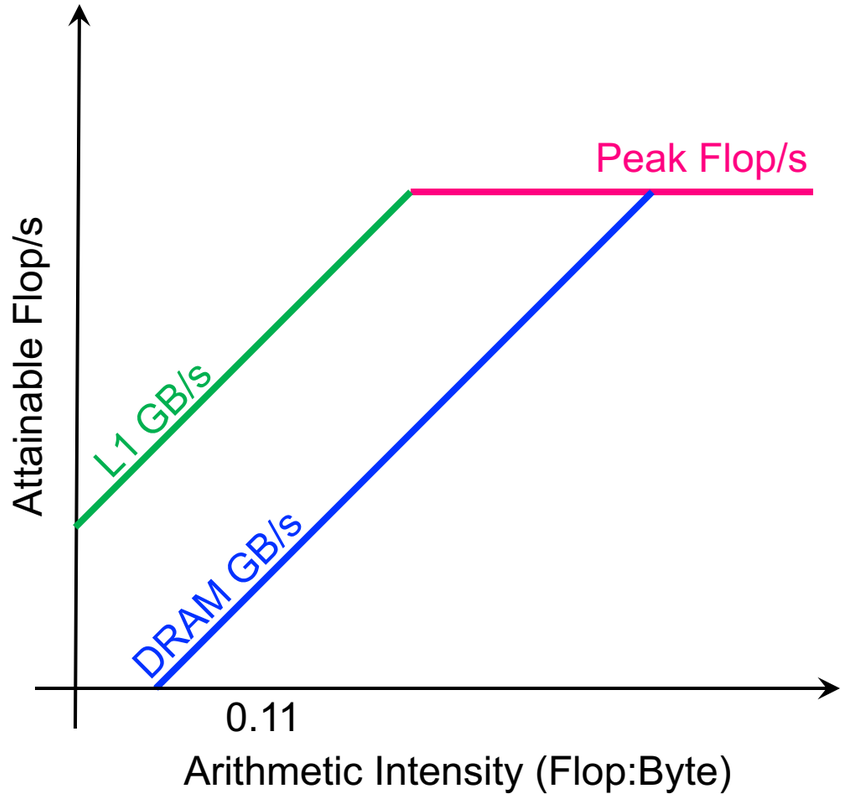
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk          ]
                 + old[ijk-1       ]
                 + old[ijk+1       ]
                 + old[ijk-jStride]
                 + old[ijk+jStride]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

Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

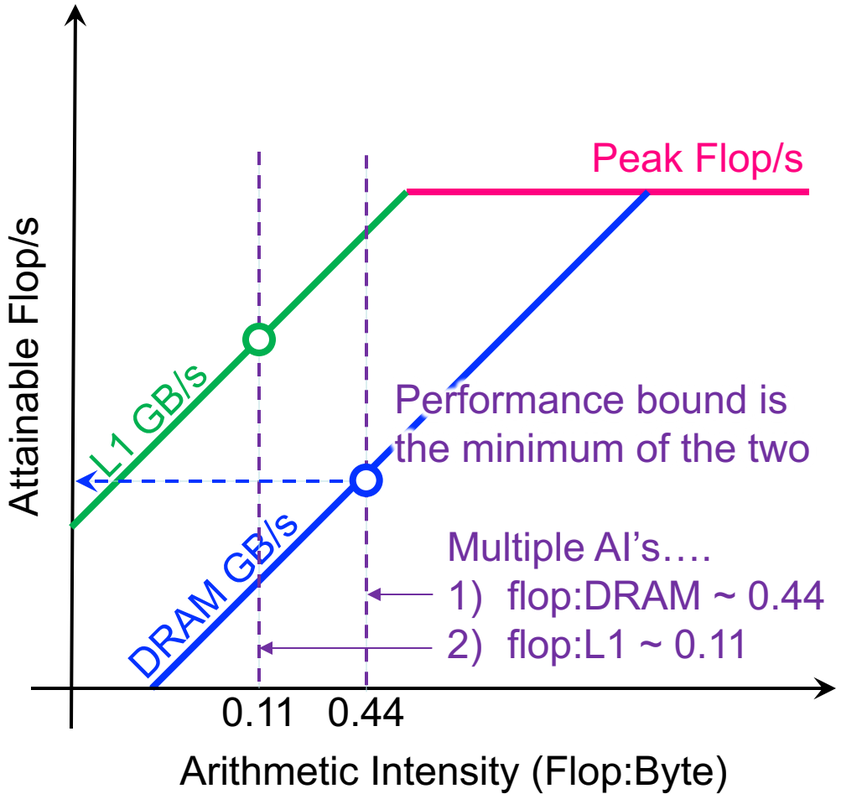


Cache-Aware Roofline

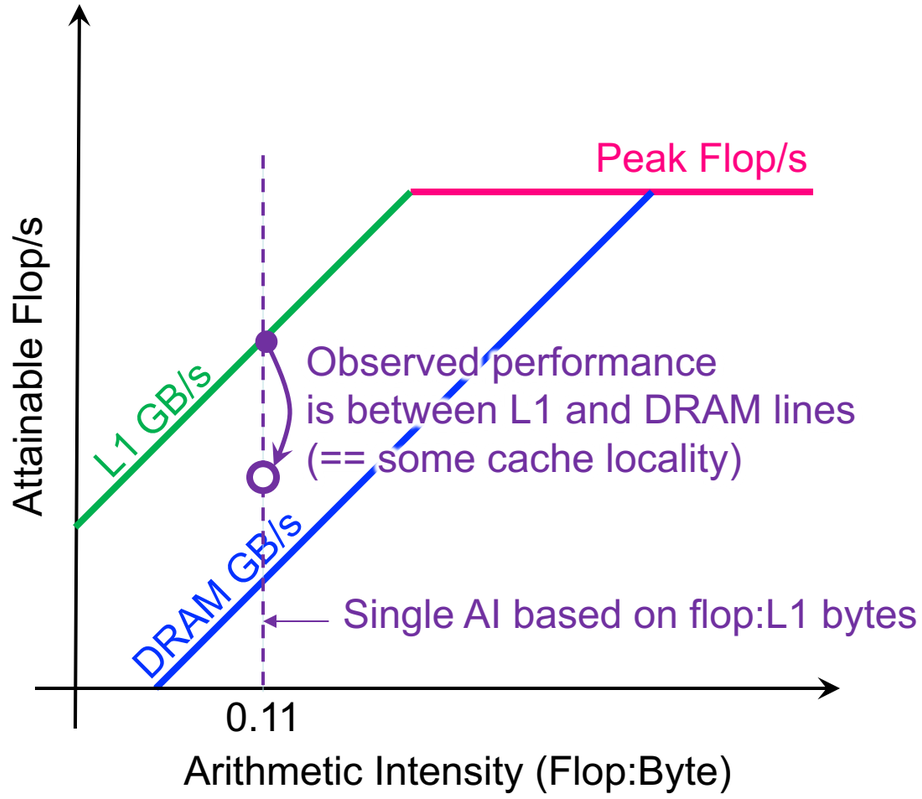


Example: 7-point Stencil (Small Problem)

Hierarchical Roofline

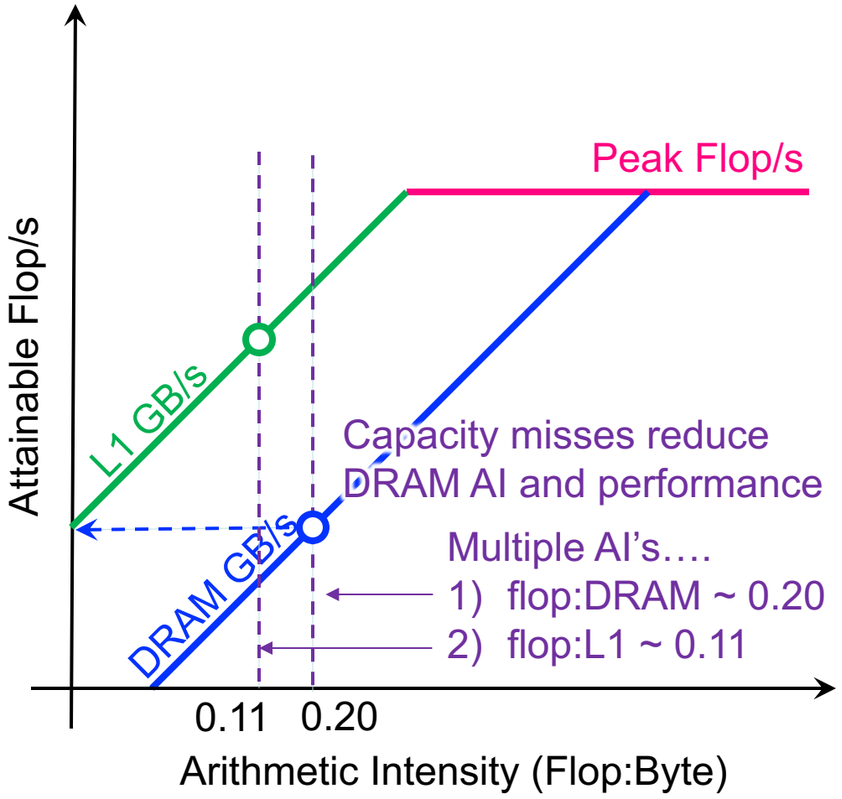


Cache-Aware Roofline

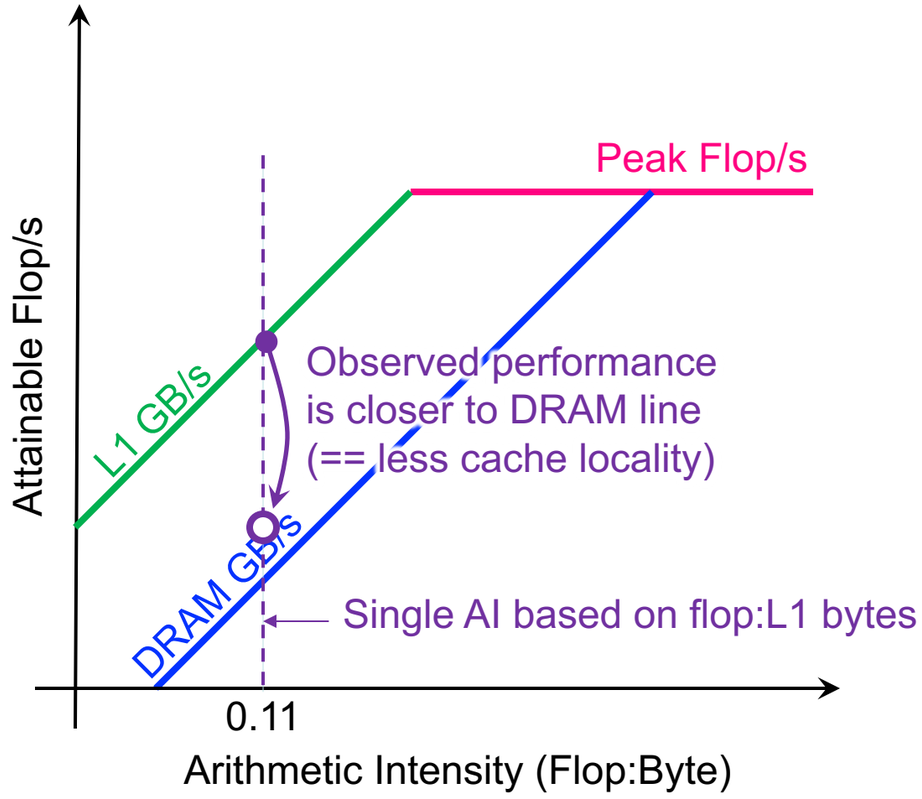


Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

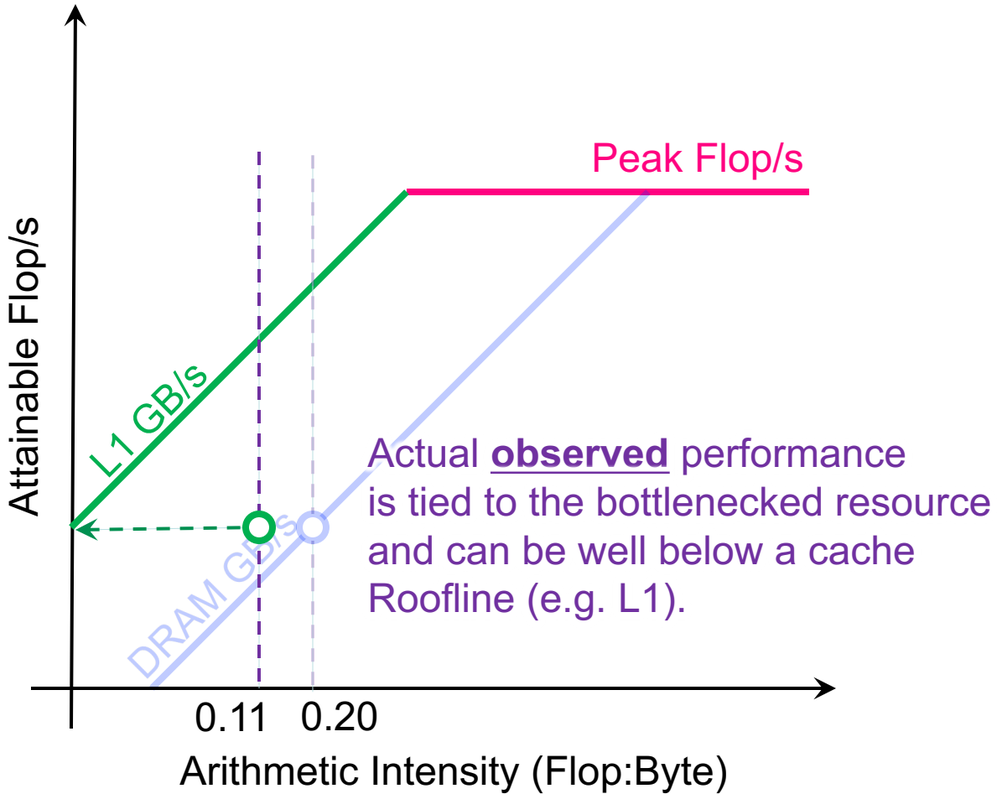


Cache-Aware Roofline

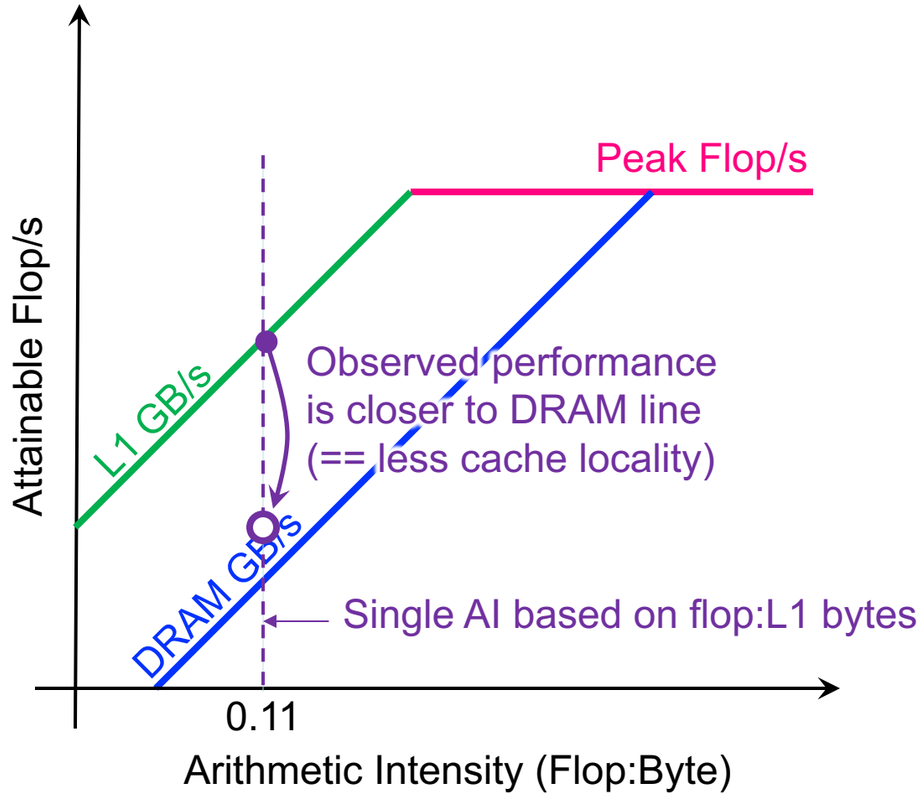


Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline

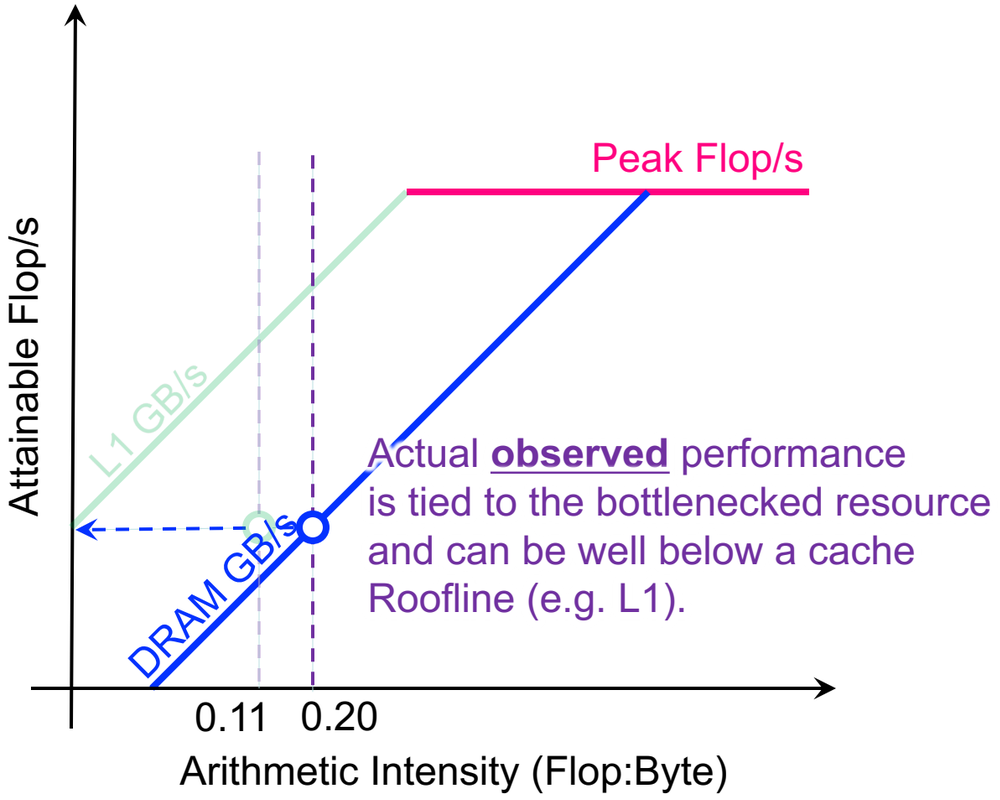


Cache-Aware Roofline



Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Cache-Aware Roofline