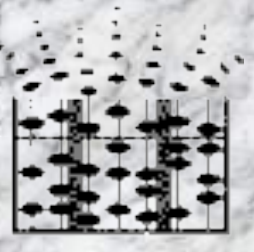
CIS 631
Parallel Processing

Lecture 3: Parallel Performance Modeling of Architectures and Algorithms

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon



Acknowledgements

- Portions of the lectures slides were adopted from:
  - A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing," 2003.
  - G. Chen, "Parallel Programming," Rensselaer Polytechnic Institute, 2002.
  - S. William, "The Roofline Model," Performance Tuning of Scientific Applications, D. Bailey, R. Lucas, and S. Williams (Eds.), CRC Press, 2011.

Logistics

- Get ACISS account
- Term paper announcement
- Exercise 1 announcement

Outline

- Performance scalability
- Analytical performance measures
- Amdahl's law and Gustafson's law
- Parallel execution models
- Isoefficiency
- Roofline model

Causes of Performance Loss

- If each processor is rated at *k* MFLOPS and there are *p* processors, shouldn't we see *k*p* MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system or problem size) can change conditions
- Need to understand *performance space*

Embarrassingly Parallel Computations

- An *embarrassingly parallel* computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes T time sequentially, there is the potential to achieve T/P time running in parallel with P processors
  - What would cause this not to be the case always?

Performance Issues

- Sequential Performance
- Critical Paths
- Bottlenecks
- Communication Performance
  - Overhead and grainsize
  - Too many messages
  - Global Synchronization
- Algorithmic overhead
- Load imbalance
- Speculative Loss
- …

Why Aren't Parallel Applications Scalable?

- Critical Paths
  - Dependencies between computations spread across processors
- Bottlenecks
  - One processor holds things up
- Algorithmic overhead
  - Some things just take more effort to do in parallel
- Communication overhead
  - Spending increasing proportion of time on communication
- Load Imbalance
  - Makes all processor wait for the "slowest" one
  - Dynamic behavior
- Speculative loss
  - Do A and B in parallel, but B is ultimately not needed

Critical Paths

- Long chain of dependence
  - Main limitation on performance
  - Resistance to performance improvement
- Diagnostic
  - Performance stagnates to a (relatively) fixed value
  - Critical path analysis
- Solution
  - Eliminate long chains if possible
  - Shorten chains by removing work from critical path

Bottlenecks

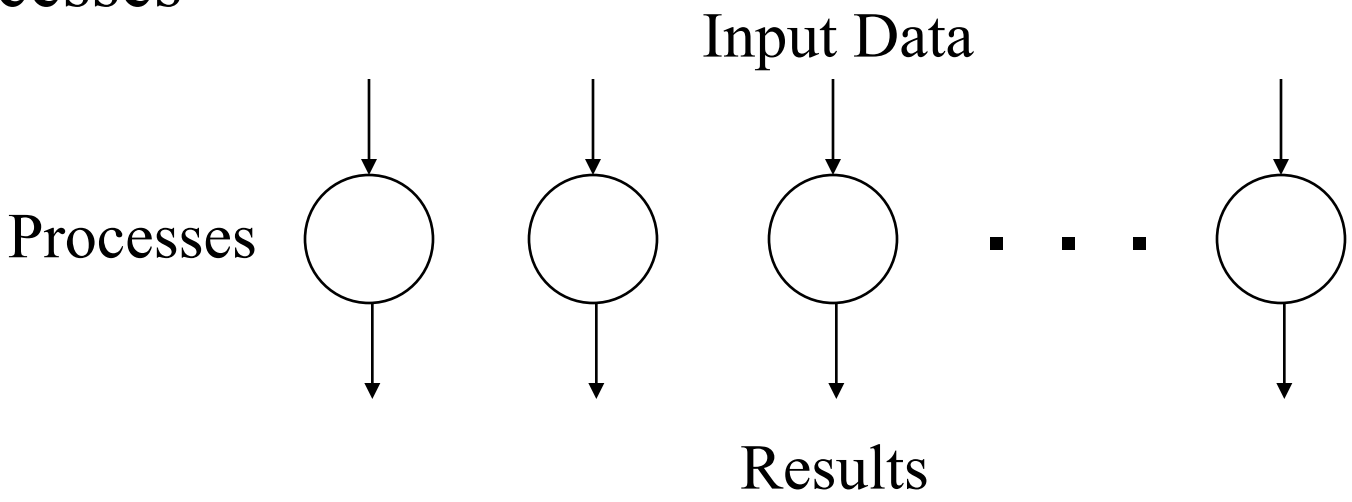
- How to detect?
  - One processor A is busy while others wait
  - Data dependency on the result produced by A
- Typical situations:
  - N-to-1 reduction / computation / 1-to-N broadcast
  - One processor assigning job in response to requests
- Solution techniques:
  - More efficient communication
  - Hierarchical schemes for master slave
- Program may not show ill effects for a long time
- Shows up when scaling

Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce addition operations (Why?)
  - *Parallel overhead*
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it doesn't parallelize well (e.g., not scalable)
- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms
- Performance is the rub
  - Are you achieving better parallel performance?
  - Must compare with the best sequential algorithm

Embarrassingly Parallel Computations

- No or very little communication between processes
- Each process can do its tasks without any interaction with other processes



- Examples
  - Numerical integration
  - Mandelbrot set
  - Monte Carlo methods

Mandelbrot Set

- The Mandelbrot set is a set of points in a complex plane that are "quasi-stable" (will increase and decrease but not exceed some limit) when computed by iterating a function:

$$z_{k+1} = z_k^2 + c$$

  - z_k is the kth iteration of the complex number $z=a+b\boldsymbol{i}$
  - z_{k+1} is the $(k+1)$th value of z
  - c is a complex number giving the position of the point in the complex plane
  - If a point has coordinates (x,y), then $c=x+y\boldsymbol{i}$

Mandelbrot Set

- Computation of a single pixel:

$$z_{k+1} = z_k^{\,2} + c$$

$$z_{k+1} = (a_k + b_k i)^2 + (x + yi)$$

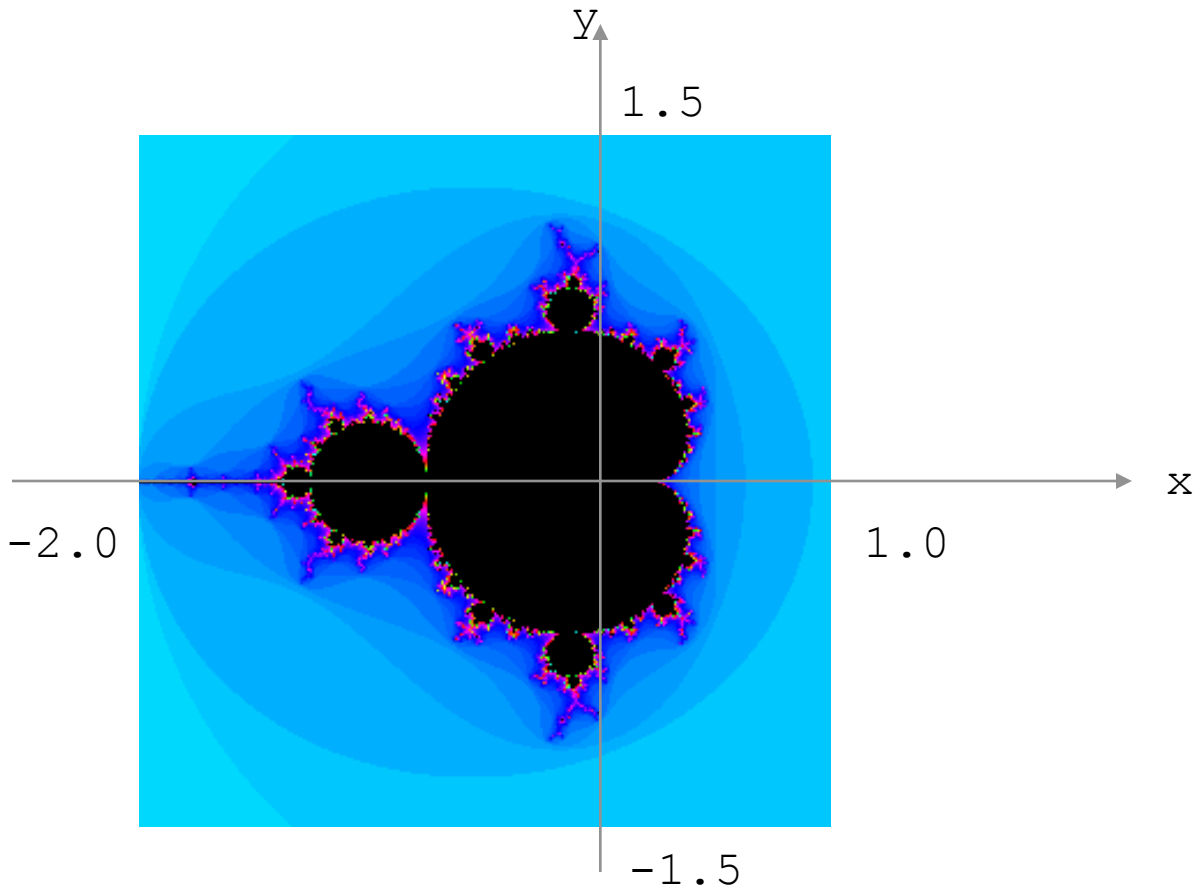
$$= (a_k^{\,2} - b_k^{\,2} + x) + (2a_k b_k + y)i$$

- Initial value of z is 0
- Iterations are continued until the magnitude of z is greater than 2 (which indicates that eventually z will become infinite) or the number of iterations reaches a threshold
- The magnitude of z is given by

$$|z| = |a + bi| = \sqrt{a^2 + b^2}$$

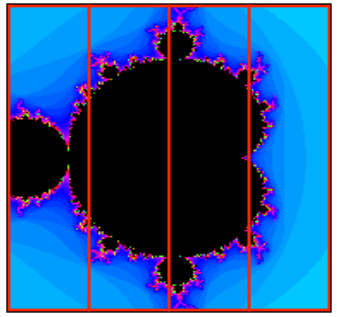
Mandelbrot Visualization

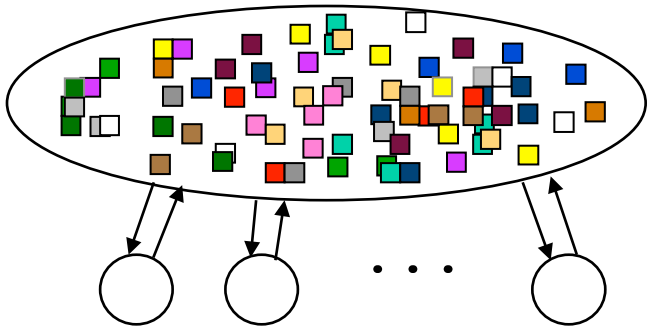
- Black points do not go to infinity
- Colors are added to the points that are not in the set



Parallelizing Mandelbrot Computation

- Mandelbrot set is embarrassingly parallel – computation of any two pixels is completely independent
- The numbers of iterations (execution times) for the computation on different pixels are different?
- Parallelization strategies:
  - Block partitioning
    - mapping greatly affects performance
  - Dynamic assignment



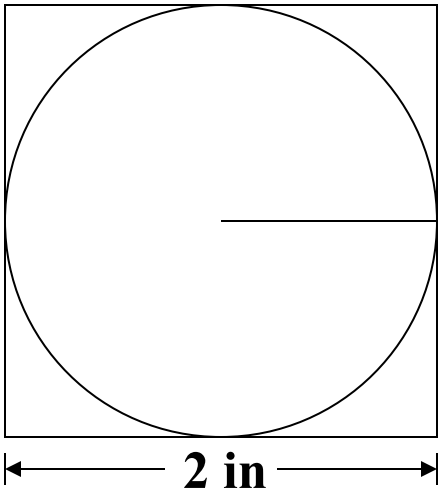


Monte Carlo Methods

- Monte Carlo methods
  - Based on the use of random selections in calculations leading to the solution of numerical and physical problems
  - Similarity of statistical simulation to games of chance
  - Also referred to as Monte Carlo simulation
- Not all simulations involving the use of random number are Monte Carlo simulation
  - Only those in which the passage of time plays no substantial role are
- Calculation on different points are independent
- It is embarrassingly parallel
- Need to generate independent random number sequences

Calculating π with Monte Carlo

- Consider a circle of unit radius
- Place circle inside a square box with length of side 2

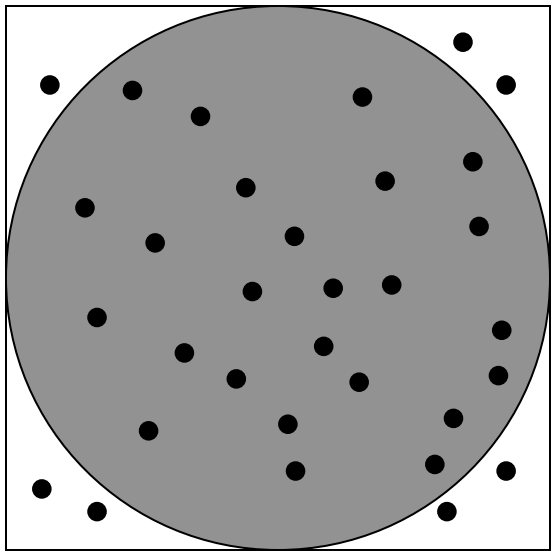


- The ratio of the circle area to the area of the square is:

$$\frac{\pi * 1 * 1}{2 * 2} = \frac{\pi}{4}$$

Monte Carlo Calculation of π

- Randomly choose a number of points in the square
- For each point *p*, determine if *p* is inside the circle
- The ratio of points in the circle to points in the square will give an approximation of π/4



Euler's Conjecture

- *It is impossible to exhibit three fourth powers whose sum is a fourth power, four fifth powers whose sum is a fifth power, and similarly for higher powers, Euler, 1769*

$$x^n = \sum_{i=1}^{n-1} y_i^n \, (n \geq 4)$$

- Is in fact a generalized Fermat's Last Theorem
  - Proved by Wiles in 1994
- The first counterexample was found by Lander and Parkin in 1966
  - $144^5 = 133^5+110^5+84^5+27^5$
- A counterexample for the fourth power was found by Elkies, 1988
  - $422481^4 = 414560^4+217519^4+95800^4$
- No counterexample known for n>5

EulerNet

- http://euler.free.fr/
- The main goal is to find a sixth power that is equal to the sum of five sixth powers
- Each participant downloads a small program that will run in the background when the CPU is idle
- The program acquires a range of numbers that need to be checked from the EulerNet server
- Solutions will be reported to the server once the Internet connection is available
- Distributed Computing
  - The computational problem to be solved must be embarrassingly parallel

Performance Metrics

- T_1 is the execution time on a single processor system
- T_p is the execution time on a p processor system
- $S(p)$ (S_p) is the *speedup*

$$S(p) = \frac{T_1}{T_p}$$

- $E(p)$ (E_p) is the *efficiency*

$$Efficiency = \frac{S_p}{p}$$

- $Cost(p)$ (C_p) is the *cost*

$$Cost = p \times T_p$$

Performance Metrics

- Parallel algorithm is *cost-optimal*
  - parallel cost = sequential time
  - $C_p = T_1$
  - $E_p = 100\%$

- Critical when down-scaling
  - Parallel algorithm may become slower than sequential

Amdahl's Law (Fixed Problem Size Speedup)

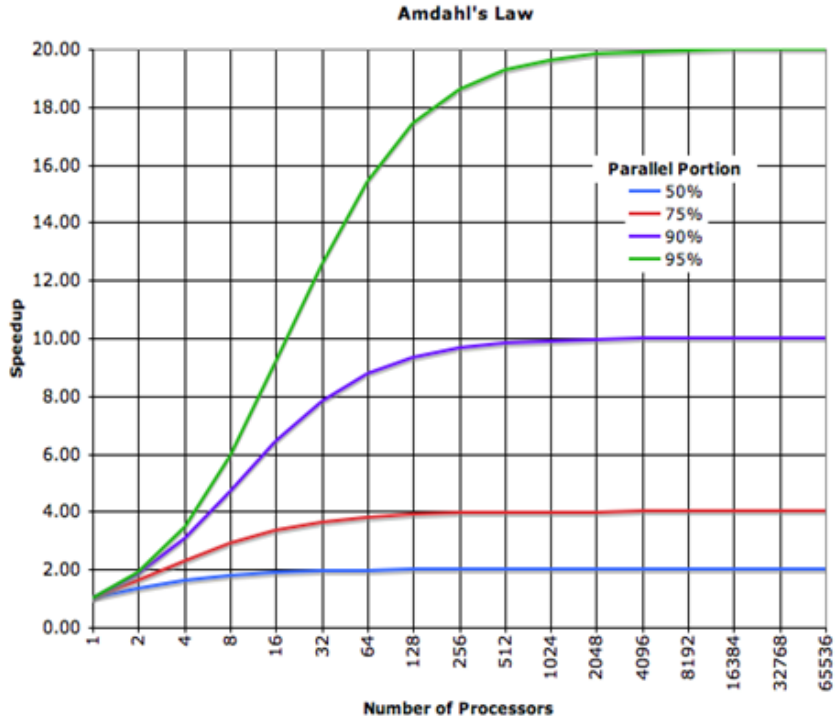
- Let f be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let T_1 be the execution time on 1 processor
- Let T_p be the execution time on p processors
- S_p is the speedup

$$S_p = T_1 / T_p$$
$$= T_1 / (f * T_1 + (1-f) * T_1 / p))$$
$$= 1 / (f + (1-f)/p))$$

- As $p \rightarrow \infty$

$$S_p = 1 / f$$

- Uhh, this is not good …



Performance and Scalability

- Evaluation
  - Sequential runtime is a function of
    - problem size and architecture
  - Parallel runtime is a function of
    - problem size and parallel architecture
    - # processors
  - Parallel performance affected by
    - algorithm + architecture
- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional with respect to the number of processors
- *Strong scaling*
  - problem size is fixed

Gustafson's Law (Scaled Speedup)

- Often interested in running larger problems when scaling
- Problem size is determined by constraint on parallel time
- Assume parallel time is kept constant
  - $T_p = C = (f + (1-f)) * C$
  - f_{seq} is the fraction of T_p spent in sequential execution
  - f_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
  - Let $C=1$, then $T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p\text{-}1)f_{par}$
- What is the speedup in this case?
  - $S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p\text{-}1)f_{par}$
- Scale the problem size as increase number of processors!

Scalability

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is parallel efficiency retained as problem size increases?
- Apply performance metrics
- *Weak scaling*
  - problem size can increase

P and NP

- *P*: solved in polynomial time $n^{O(1)}$
- *NP*: verified in polynomial time $n^{O(1)}$
- Every known *NP* problem can be solved in exponential time $n^{O(n)}$
- A problem is *NP*-complete if:
  - It is an *NP* problem, and
  - Every problem in *NP* can be polynomially reduced into this problem

Parallel Feasibility

- A problem is ***feasible*** if it can be solved by a parallel algorithm with worst case time complexity $n^{O(1)}$ and processor complexity $n^{O(1)}$
- A problem is ***highly parallel*** if it can be solved by a parallel algorithm worst case time complexity $(\log n)^{O(1)}$ and processor complexity $n^{O(1)}$
- A parallel algorithm is ***inherently sequential*** if it is feasible, but has no feasible highly parallel algorithm for its solution
- The class of feasible parallel problems is equivalent to the class of *P*

NC and P-Complete

- *NC* (Nick's Class) is the class of highly parallel problems
- There is a general belief, but not a proof, that P≠NC
- A problem *L* is said to be *P-complete* if
  - $L \in P$, and
  - Every other problem in *P* can be transformed to *L* in polylogarithmic $(\log n)^{O(1)}$ parallel time using $n^{O(1)}$ processors
- Such a transformation is said to be an *NC-reduction*
- A P-complete problem is inherently sequential
- If we could find (unlikely) an $L \in P\text{-}Complete$ and $L \in NC$, the it would follow that $P=NC$

Parallel Computation Thesis

- Polynomial sequential space is related to polylogarithmic parallel time
- In other words, what can be computed in $n^{O(1)}$ sequential space can be computed in $(\log n)^{O(1)}$ parallel time, and vice versa
- If a problem p_1 can be transformed to the problem p_2 using polynomial space, then the transformation is also possible using an *NC* reduction

An NC Problem

- Sum
  - Given: natural numbers $a_1, a_2, \ldots, a_n$
  - Problem: what is $a_1 + a_2 + \ldots + a_n$?
- Sequential time complexity
  - $O(n)$
- Parallel time complexity
  - n processors: $O(\log n)$
- $S_p = O(n/\log n)$
- $E_p = O(1/\log n)$
- Cost optimal

Another NC Problem

- Matrix multiplication
  - Given: two *nxn* matrices *A* and *B*
  - Problem: what is *AxB*?
- Sequential time complexity
  - $O(n^3)$
- Parallel time complexity
  - *n* processors: $O(n^2)$
  - n^2 processors: $O(n)$
  - n^3 processors: $O(\log n)$

Question

- For a problem with sequential time complexity $T(n)=n^{O(1)}$
  - If the problem can be solved cost-optimally using n processors, is it highly parallel?
  - If the problem can be solved cost-optimally using $T(n)$ processors, is it highly parallel?

Review

- A problem in P is highly parallel (NC) if it runs in polylogarithmic time $(\log n)^{O(1)}$ on polynomial $n^{O(1)}$ processors; otherwise, it is inherently sequential
- A P problem is P-complete if every problem in P can be transformed into this problem using an NC-reduction
- A P-complete problem is very unlikely to be highly parallel

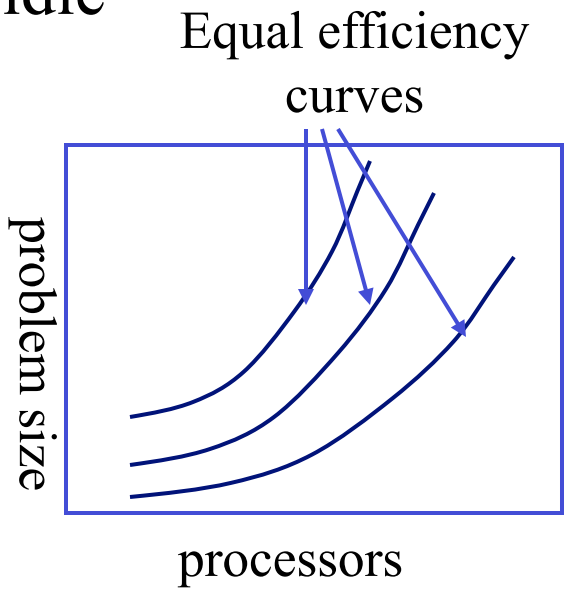
Major Analytical / Theoretical Techniques

- Typically involves simple algebraic formulas, and ratios
  - Typical variables are:
    - data size (N), number of processors (P), machine constants
  - Model performance of individual operations, components, algorithms in terms of the above
    - Be careful to characterize variations across processors, and model them with (typically) max operators
  - Constants are important in practical parallel computing
    - Be wary of asymptotic analysis: use it, but carefully
- Scalability analysis:
  - Isoefficiency

(See Kumar paper online and chapter slides)

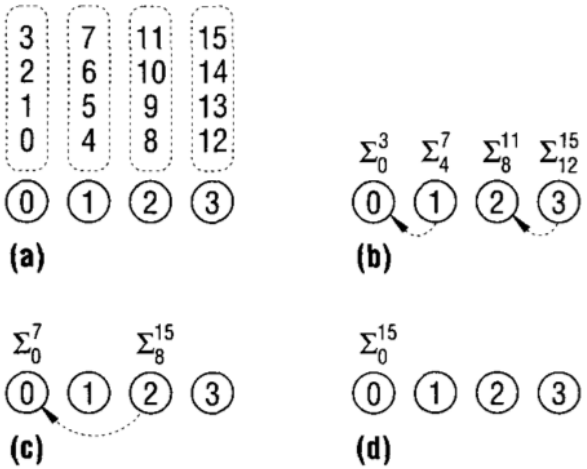
Isoefficiency

- Quantify scalability
- How much increase in problem size is needed to retain the same efficiency on a larger machine?
- Efficiency
  - T1 / (p * Tp)
  - Tp = computation + communication + idle
- Isoefficiency
  - Equation for equal-efficiency curves
  - If no solution
    - problem is not scalable in the sense defined by isoefficiency



Scalability of Adding n Numbers

- Scalability of a parallel system is a measure of its capacity to increase speedup with more processors
- Adding n numbers on p processors with strip partition:

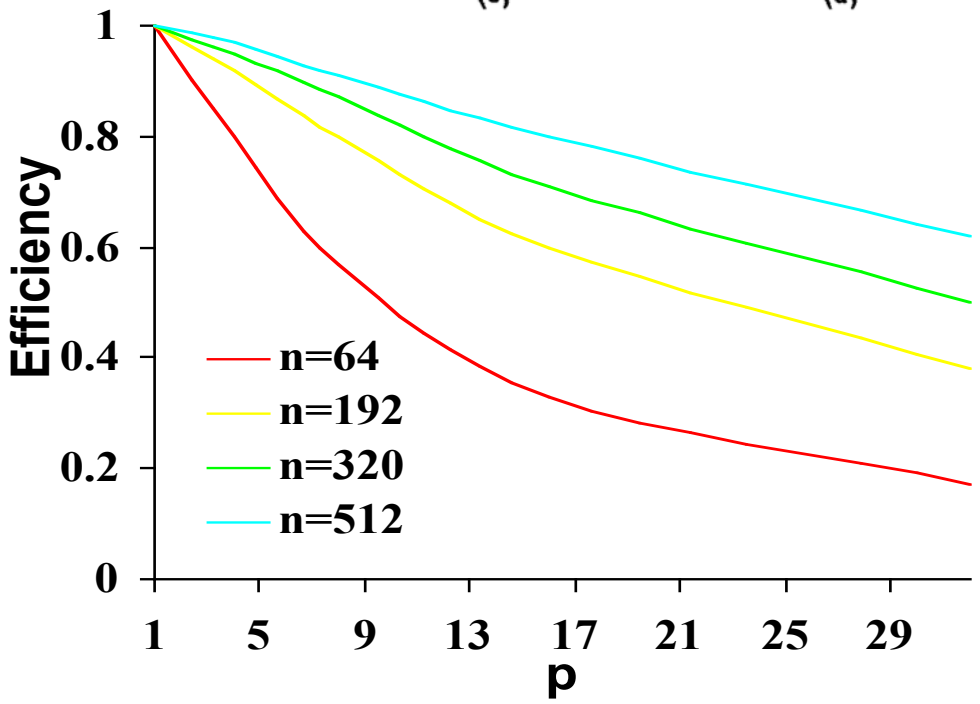


$$T_{par} = \frac{n}{p} - 1 + 2\log p$$

$$Speedup = \frac{n-1}{\frac{n}{p} - 1 + 2\log p}$$

$$\approx \frac{n}{\frac{n}{p} + 2\log p}$$

$$Efficiency = \frac{S}{p} = \frac{n}{n + 2p\log p}$$



Problem Size

- Informally, problem size is expressed as a parameter of the input size
  - How do we define the problem size for an *nxn* matrix?
- A consistent definition of the size of the problem is the total number of basic operations required to solve the problem, or T_{seq}
- Also refer to problem size as "work", denoted it by W:

$$W = T_{seq}$$

Overhead Function

- The overhead function of a parallel system is defined as the part of the cost that is not incurred by the best serial algorithm
- Denoted by T_O, it is a function of W and p

$$T_O(W,p) = pT_{par} - W \qquad (pT_{par} \text{ includes overhead})$$

$$T_O(W,p) + W = pT_{par}$$

- Overhead function of adding n numbers on p processor

$$T_O(W,p) = p(n/p - 1 + 2\log(p)) - (n-1) \approx 2p\log(p)$$

Isoefficiency Function

- With a fixed efficiency, W can be expressed as a function of p

$$T_{par} = \frac{W + T_O(W, p)}{p} \qquad W = T_{seq}$$

$$Speedup = \frac{W}{T_{par}} = \frac{Wp}{W + T_O(W, p)}$$

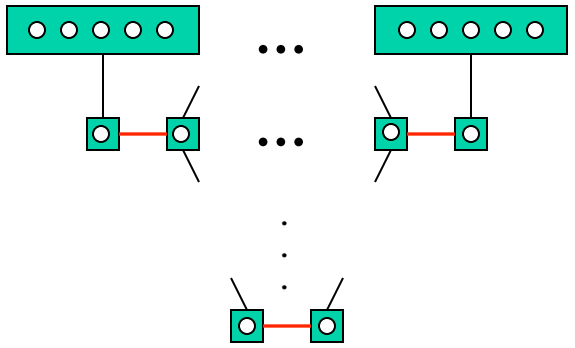
$$Efficiency = \frac{S}{p} = \frac{W}{W + T_O(W, p)} = \frac{1}{1 + \frac{T_O(W, p)}{W}}$$

$$E = \frac{1}{1 + \frac{T_O(W, p)}{W}} \rightarrow \frac{T_O(W, p)}{W} = \frac{1 - E}{E}$$

$$W = \frac{E}{1 - E} T_O(W, p) = K T_O(W, p) \quad \text{Isoefficiency Function}$$

Isoefficiency Function of Adding n Numbers

- Overhead function:
  - $T_O(W,p) = pT_{par} - W = 2plog(p)$
- Isoefficiency function:
  - $W = K*2plog(p)$
- If p doubles, W needs also to be doubled to roughly maintain the same efficiency



More Complex Isoefficiency Functions

- A typical overhead function T_O can have several distinct terms of different orders of magnitude with respect to both p and W
- We can balance W against each term of T_O and compute the respective isoefficiency functions for individual terms
  - Keep only the term that requires the highest grow rate with respect to p
  - This is the asymptotic isoefficiency function

Isoefficiency

- Consider a parallel system with an overhead function

$$T_O = p^{3/2} + p^{3/4} W^{3/4}$$

- Using only the first term

$$W = Kp^{3/2}$$

- Using only the second term

$$W = Kp^{3/4} W^{3/4}$$

$$W^{1/4} = Kp^{3/4}$$

$$W = K^4 p^3$$

- K^4p^3 gives the overall asymptotic isoefficiency function

Parallel Computation Models

- PRAM (parallel RAM)
- BSP
- LogP
- Roofline

PRAM

- Parallel Random Access Machine (PRAM)
- Shared-memory multiprocessor model
- Unlimited number of processors
  - Unlimited local memory
  - Each processor knows its ID
- Unlimited shared memory
- Inputs/outputs are placed in shared memory
- Memory cells can store an arbitrarily large integer
- Each instruction takes unit time
- Instructions are sychronized across processors (SIMD)

PRAM Complexity Measures

- For each individual processor
  - *Time*: number of instructions executed
  - *Space*: number of memory cells accessed
- PRAM machine
  - *Time*: time taken by the longest running processor
  - *Hardware*: maximum number of active processors
- Technical issues
  - How processors are activated
  - How shared memory is accessed

Processor Activation

- P_0 places the number of processors (p) in the designated shared-memory cell
  - Each active P_i, where $i < p$, starts executing
  - *O(1)* time to activate
  - All processors halt when P_0 halts

- Active processors explicitly activate additional processors via FORK instructions
  - Tree-like activation
  - *O(log p)* time to activate

PRAM is a Theoretical (Unfeasible) Model

- Interconnection network between processors and memory would require a very large amount of area
- The message-routing on the interconnection network would require time proportional to network size
- Algorithm's designers can forget the communication problems and focus their attention on the parallel computation only
- There exist algorithms simulating any PRAM algorithm on bounded degree networks
- Design general algorithms for the PRAM model and simulate them on a feasible network

Classification of PRAM Models

- *EREW* (Exclusive Read Exclusive Write)
  - No concurrent read/writes to the same memory location
- *CREW* (Concurrent Read Exclusive Write)
  - Multiple processors may read from the same global memory location in the same instruction step
- *ERCW* (Exclusive Read Concurrent Write)
  - Concurrent writes allowed
- *CRCW* (Concurrent Read Concurrent Write)
  - Concurrent reads and writes allowed
- CRCW > (ERCW,CREW) > EREW

CRCW PRAM Models

- COMMON: all processors concurrently writing into the same address must be writing the same value
- ARBITRARY: if multiple processors concurrently write to the address, one of the competing processors is randomly chosen and its value is written into the register
- PRIORITY: if multiple processors concurrently write to the address, the processor with the highest priority succeeds in writing its value to the memory location
- COMBINING: the value stored is some combination of the values written, e.g., sum, min, or max
- COMMON-CRCW model most often used

Complexity of PRAM Algorithms

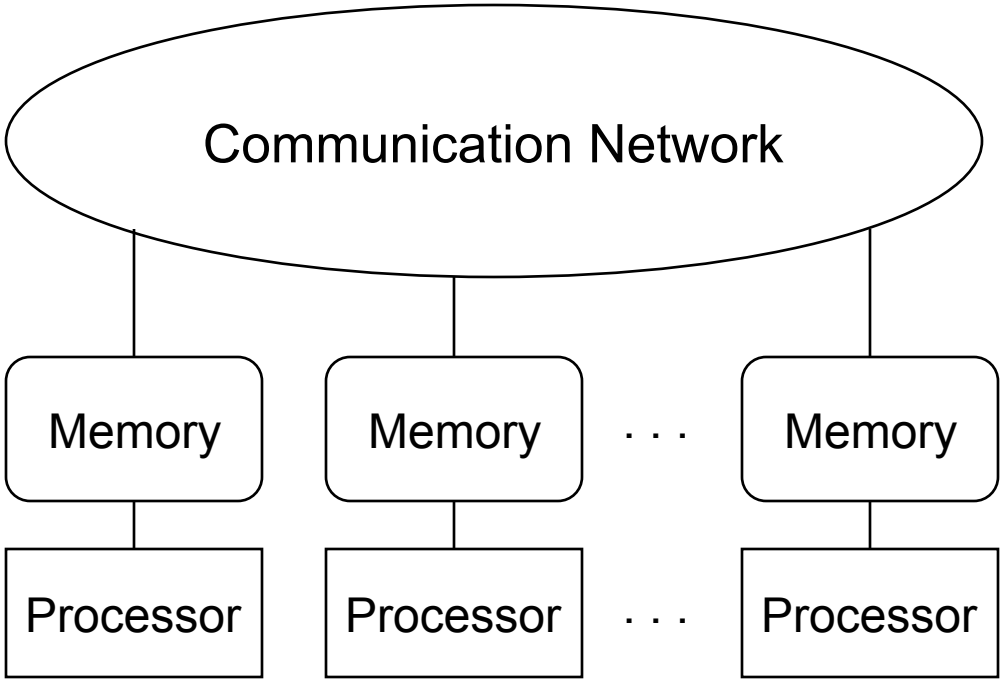
Problem / **Model**	EREW	CRCW
Search	O(log n)	O(1)
List Ranking	O(log n)	O(log n)
Prefix	O(log n)	O(log n)
Tree Ranking	O(log n)	O(log n)
Finding Minimum	O(log n)	O(1)

BSP Overview

- Bulk Synchronous Parallelism
- A parallel programming model
- Invented by Leslie Valiant at Harvard
- Enables performance prediction
- SPMD style
- Supports both direct memory access and message passing
- BSPlib is a BSP library implemented at Oxford

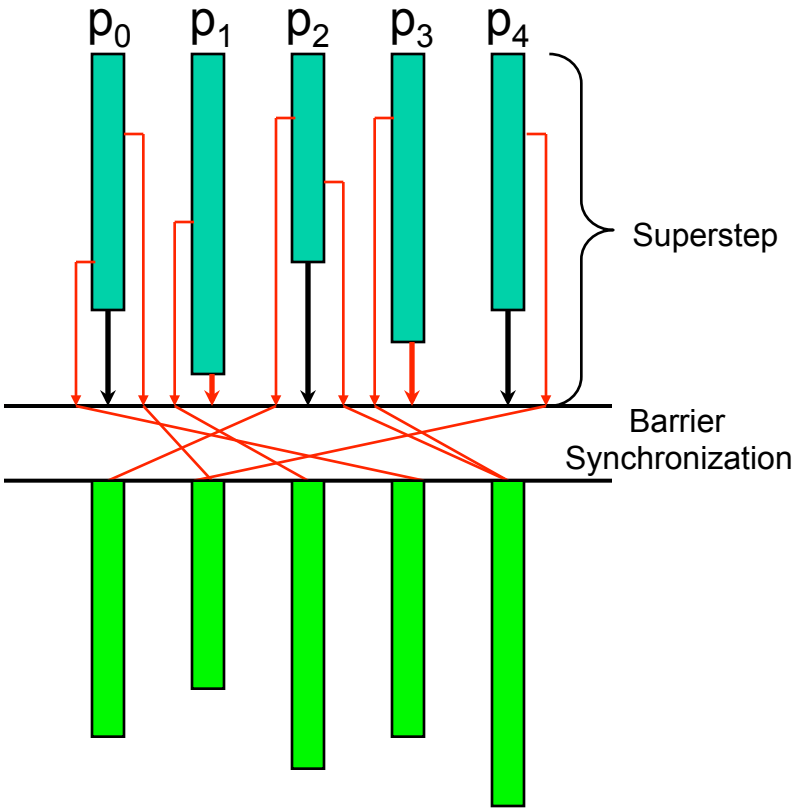
Components of BSP Computer

- A set of processor-memory pairs
- A communication point-to-point network
- A mechanism for efficient barrier synchronization of all processors



Supersteps

- A BSP computation consists of a sequence of *supersteps*
- In each superstep, processes execute computations using locally available data, and issue communication requests
- Processes synchronized at the end of the superstep, at which all communications issued have been completed



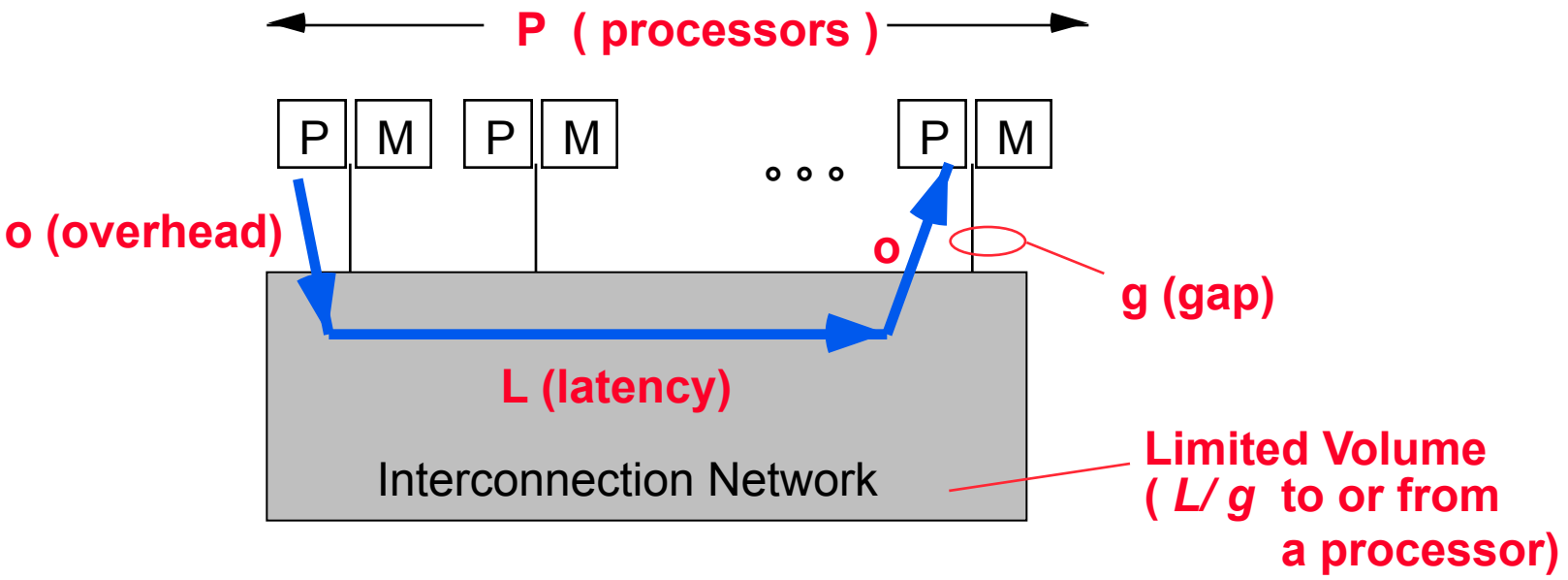
BSP Parameters

- p = number of processors
- l = barrier latency, cost of achieving barrier synchronization
- g = communication cost per word
- s = processor speed
- l, g, and s are measured in FLOPS
- Any processor sends and receives at most h messages in a single superstep (called h-relation communication)
- Time for a superstep = max number of local operations performed by any one processor + $g*h + l$

The LogP Model (Culler, Berkeley)

- Processing
  - Powerful microprocessor, large DRAM, cache => P
- Communication
  - Significant latency (100's of cycles) => L
  - Limited bandwidth (1 – 5% of memory) => g
  - Significant overhead (10's – 100's of cycles) => o
    - on both ends
    - no consensus on topology
    - should not exploit structure
  - Limited capacity
- No consensus on programming model
  - Should not enforce one

LogP



- Latency in sending a (small) mesage between modules
- overhead felt by the processor on sending or receiving message
- gap between successive sends or receives (1/BW)
- Processors

LogP "Philosophy"

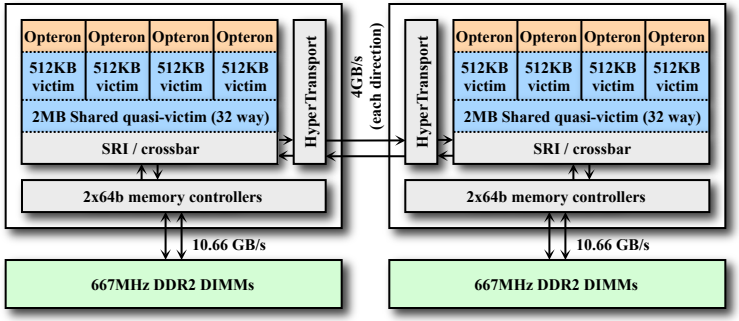
- Think about:
  - Mapping of N words onto P processors
  - Computation within a processor, its cost, and balance
  - Communication between processors, its cost, and balance
- Characterize the processor and network performance
- Do not think about what happens within the network
- This should be enough

Typical Values for g and l

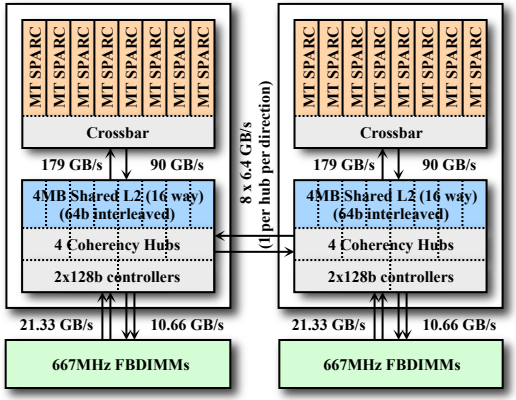
	p	g	l
Multiprocessor Sun	2-4	3	50-100
SGI Origin 2000	2-8	10-15	1000-4000
IBM-SP2	2-8	10	2000-5000
NOW (Network of Workstations)	2-8	40	5000-20000

Roofline Model (S. Williams)

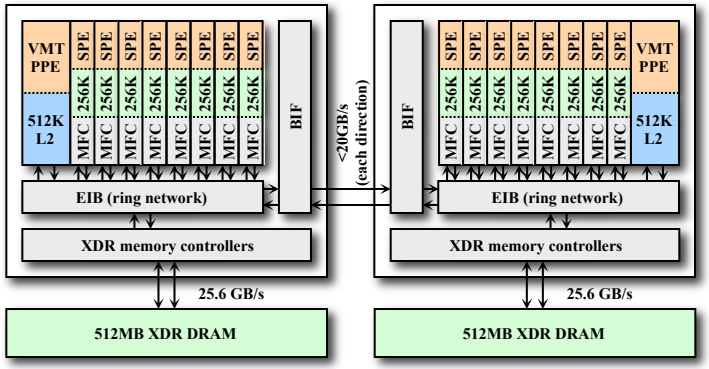
AMD Barcelona



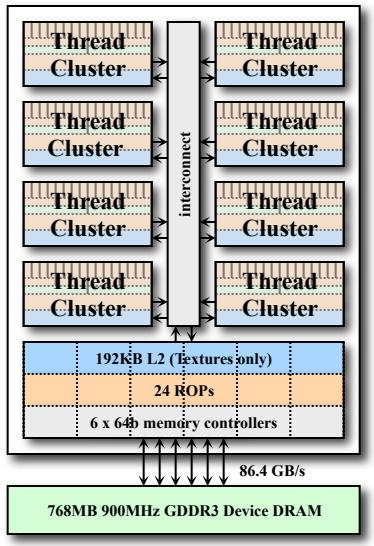
Sun Victoria Falls



IBM Cell Blade



NVIDIA G80



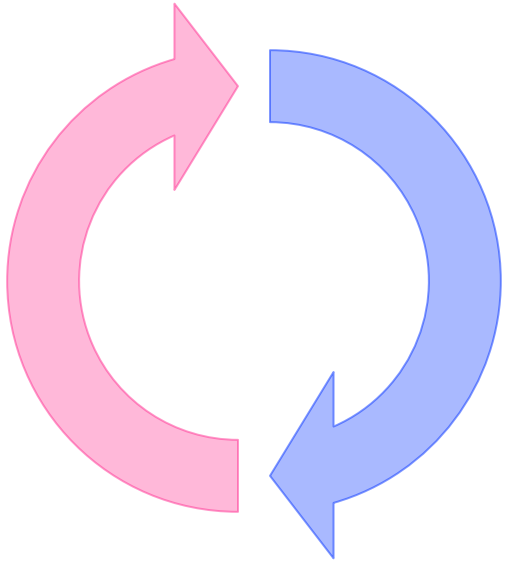
Challenges / Goals

- We have extremely varied architectures
- Moreover, the characteristics of numerical methods can vary dramatically
- The result is that performance and the benefit of optimization can vary significantly from one architecture x kernel combination to the next
- We wish to understand whether or not we have attained good performance (high fraction of a theoretical peak)
- We wish to identify performance bottlenecks and enumerate potential remediation strategies

Performance Optimization: Contending Forces

- Contending forces of device efficiency and usage/traffic
- We improve time to solution by improving throughput (efficiency) and reducing traffic

Improve Throughput
(Gflop/s, GB/s, etc…)

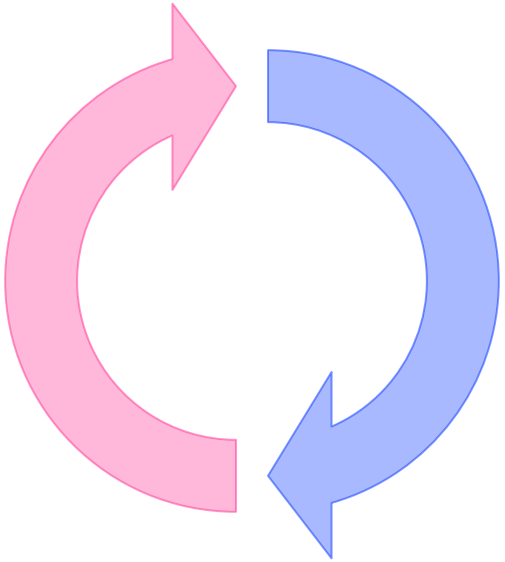


Reduce Volume of Data
(Flop's, GB's, etc…)

Performance Optimization: Contending Forces

- Contending forces of device efficiency and usage/traffic
- We improve time to solution by improving throughput (efficiency) and reducing traffic

Restructure to satisfy Little's Law



Implementation & Algorithmic Optimization

- In practice, we are willing to sacrifice one in order to improve the time to solution

Basic Throughput Quantities

- At all levels of the system (register files through networks), there are three fundamental (efficiency-oriented) quantities:
  - **Latency**
    - every operation requires time to execute

      (i.e. instruction, memory or network latency)
  - **Bandwidth**
    - # of (parallel) operations completed per cycle

      (i.e. #FPUs, DRAM, Network, etc…)
  - **Concurrency**
    - total # of operations in flight

Little's Law

- Little's Law:

  Concurrency = Latency * Bandwidth

  - or –

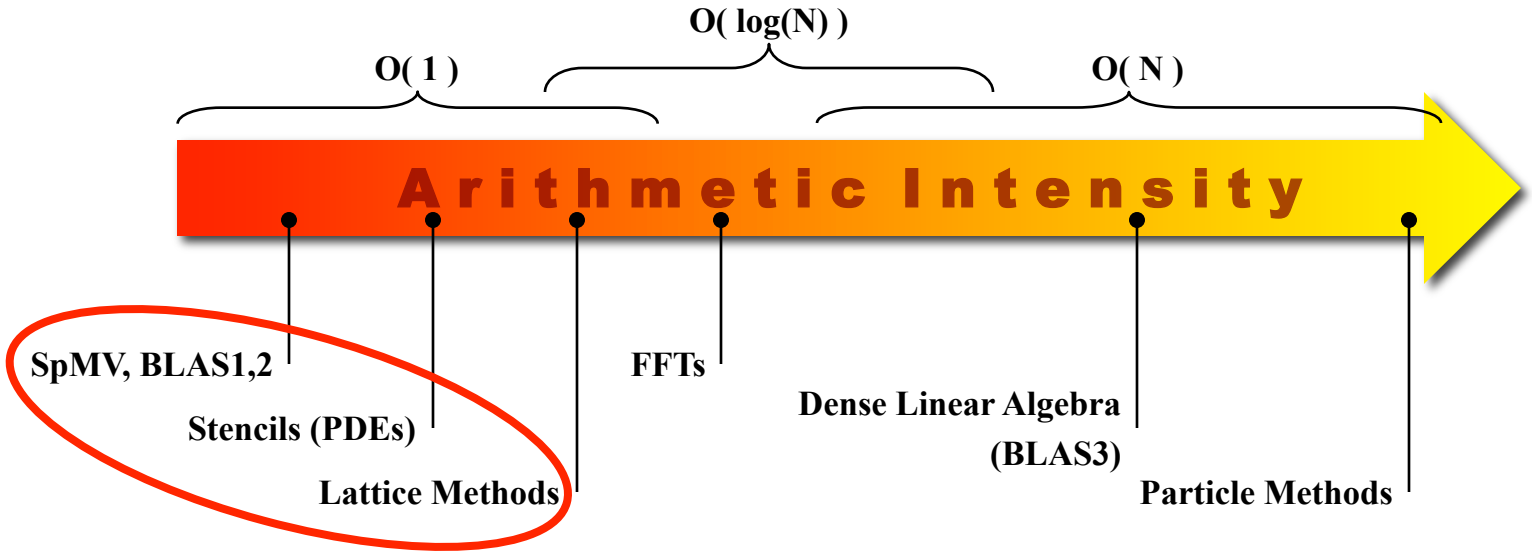
  Effective Throughput = Expressed Concurrency / Latency

- This concurrency must be filled with parallel operations
- Can not exceed peak throughput with more concurrency
  - each channel has a maximum throughput
- 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

Three Classes of Locality

- Temporal Locality
  - Reusing data (either registers or cache lines) multiple times
  - Amortizes the impact of limited bandwidth
  - **Transform loops or algorithms to maximize reuse**
- Spatial Locality
  - Data is transferred from cache to registers in words
  - However, data is transferred to the cache in 64-128Byte lines
  - Using every word in a line maximizes spatial locality
  - **Transform data structures into *structure of arrays* (SoA) layout**
- Sequential Locality
  - Many memory address patterns access cache lines sequentially
  - CPU's hardware stream prefetchers exploit this observation to hide speculatively load data to memory latency
  - **Transform loops to generate (a few) long, unit-stride accesses**

Arithmetic Intensity

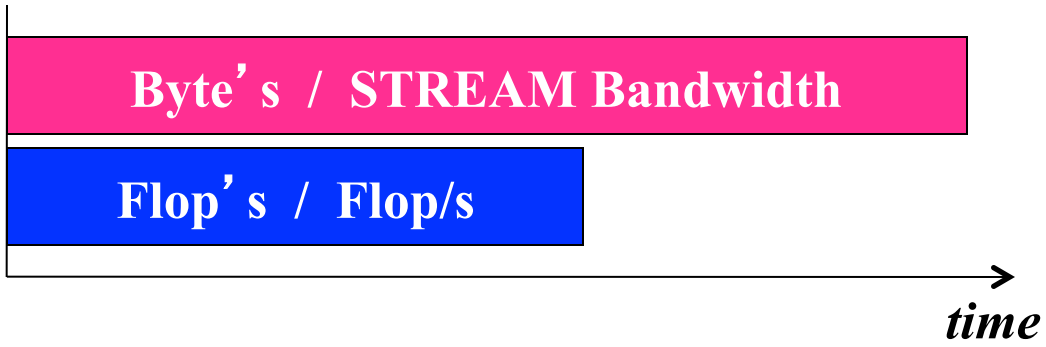


- True Arithmetic Intensity (AI)

  AI ~ Total Flops / Total DRAM Bytes
- Some HPC kernels have an arithmetic intensity that scales with problem size (increased temporal locality)
- Others have constant intensity
- Arithmetic intensity is ultimately limited by compulsory traffic
- Arithmetic intensity is diminished by conflict or capacity misses

Overlap of Communication

- Consider a simple example in which a FP kernel maintains a working set in DRAM
- We assume we can perfectly overlap computation with communication
  - Either through prefetching/DMA and/or pipelining (decoupling of communication and computation)
- Time is the maximum of the time required to transfer the data and the time required to perform the FP operations



Basic Concept

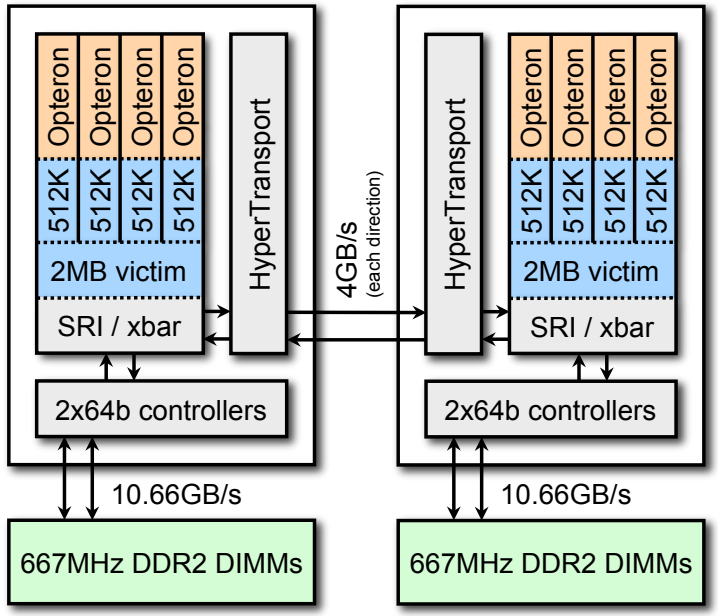
- Synthesize communication, computation, and locality into a single visually-intuitive performance figure using bound and bottleneck analysis

$$\textbf{Attainable Performance}_{ij} = \min \begin{cases} \textbf{FLOP/s with Optimizations}_{1\text{-}i} \\ \textbf{AI * Bandwidth with Optimizations}_{1\text{-}j} \end{cases}$$

- Where i and j are optimizations
- Given a kernel's arithmetic intensity (based on DRAM traffic after being filtered by the cache), programmers can inspect the figure, and bound performance
- Moreover, provides insights as to which optimizations will potentially be beneficial

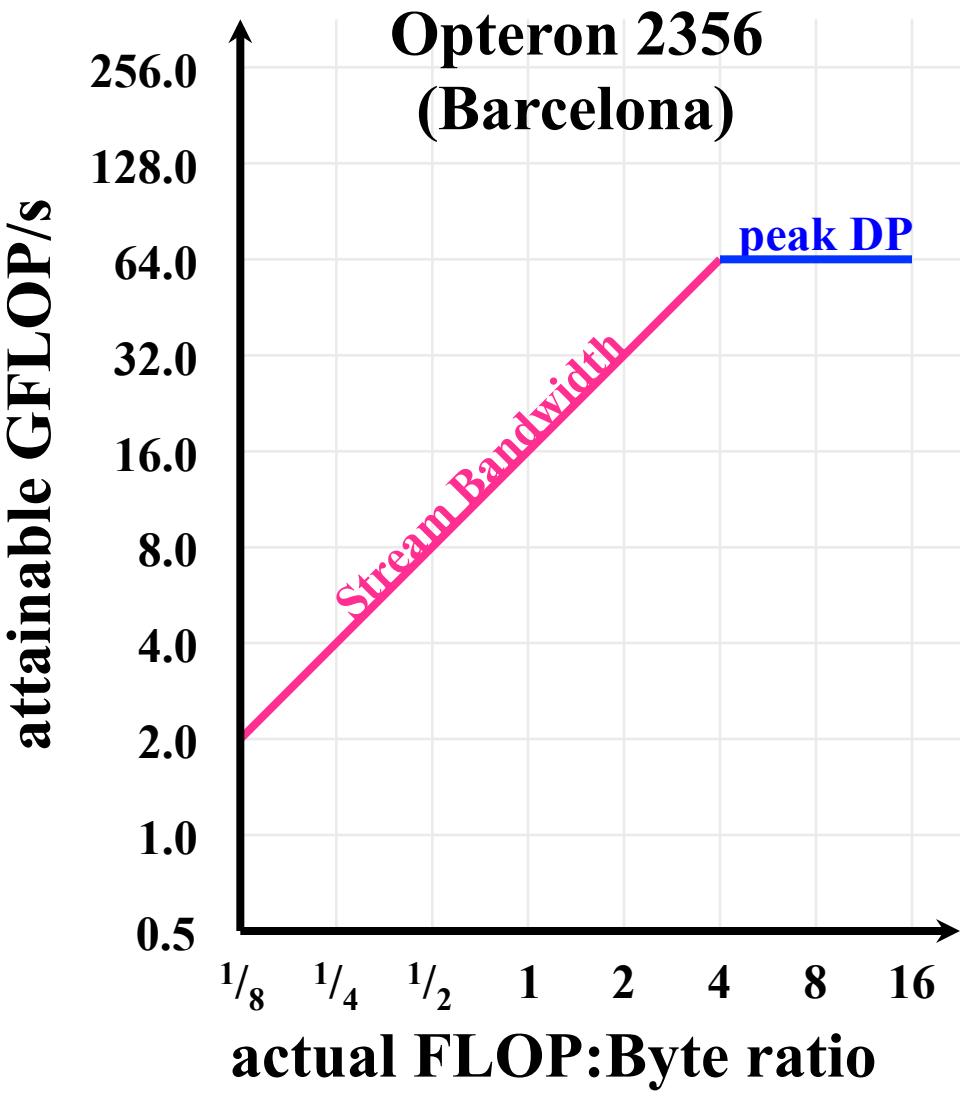
Example

- Consider the Opteron 2356:
  - Dual Socket (NUMA)
  - limited HW stream prefetchers
  - quad-core (8 total)
  - 2.3GHz
  - 2-way SIMD (DP)
  - separate FPMUL and FPADD datapaths
  - 4-cycle FP latency



- Assuming expression of parallelism is the challenge, what would the roofline model look like?

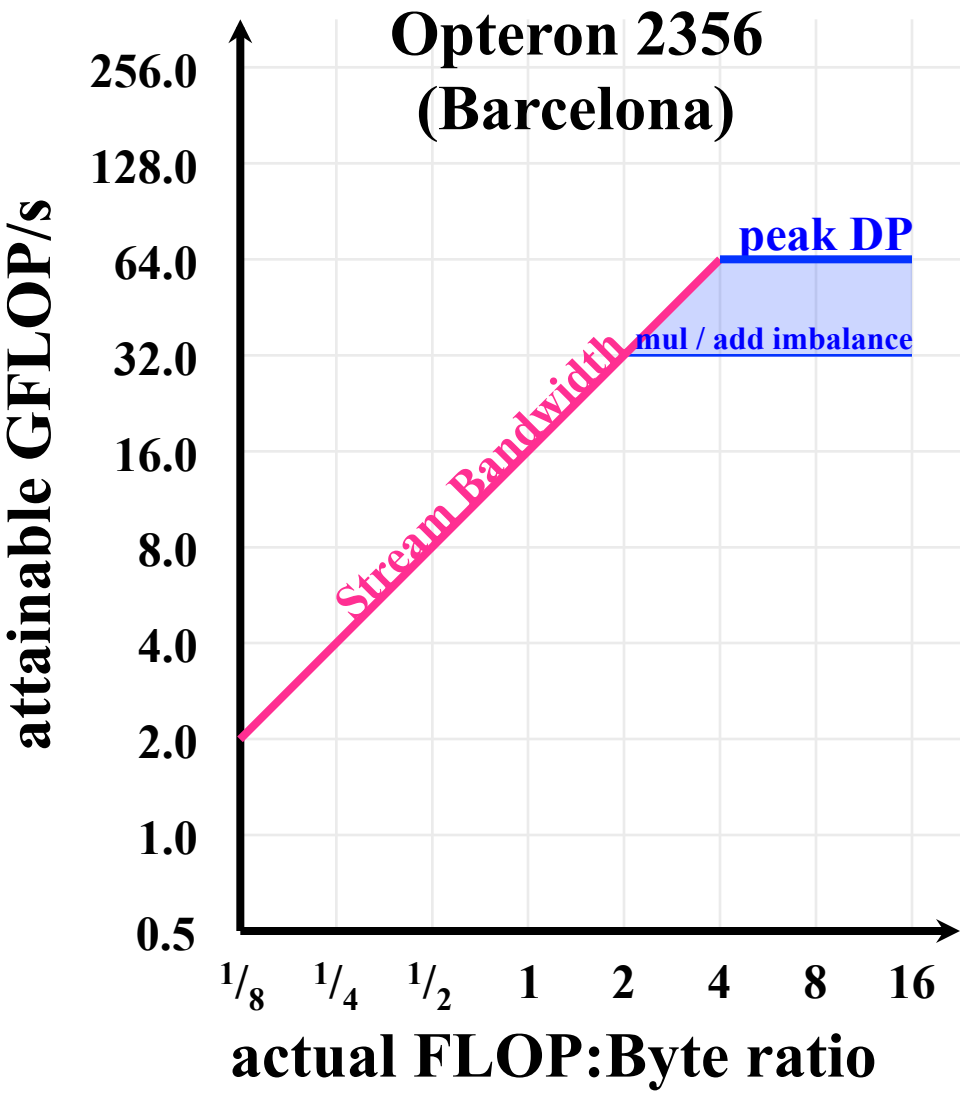
Roofline Model for Opteron 2356



- Plot on log-log scale
- Given AI, we can easily bound performance
- But architectures are much more complicated

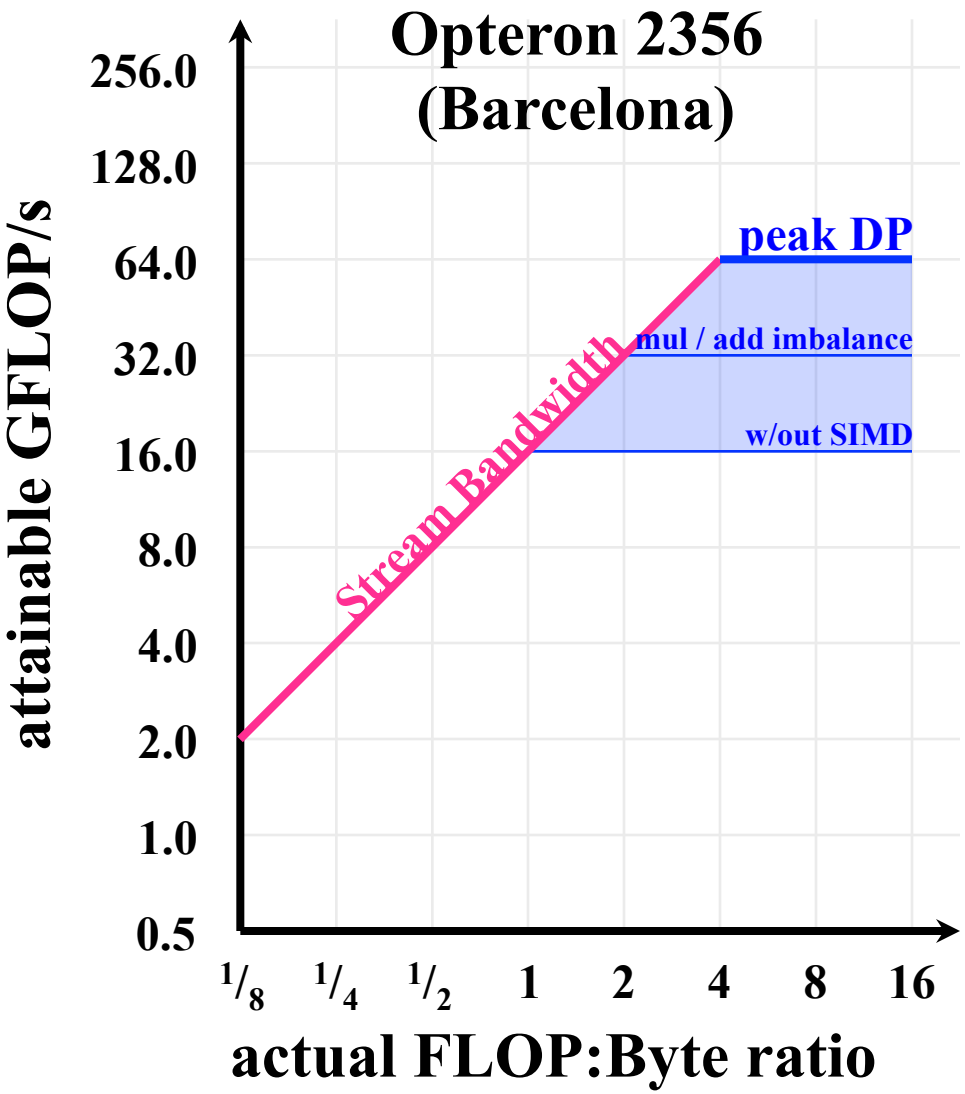
- We will bound performance as we eliminate specific forms of in-core parallelism

Roofline Model – Computational Ceilings



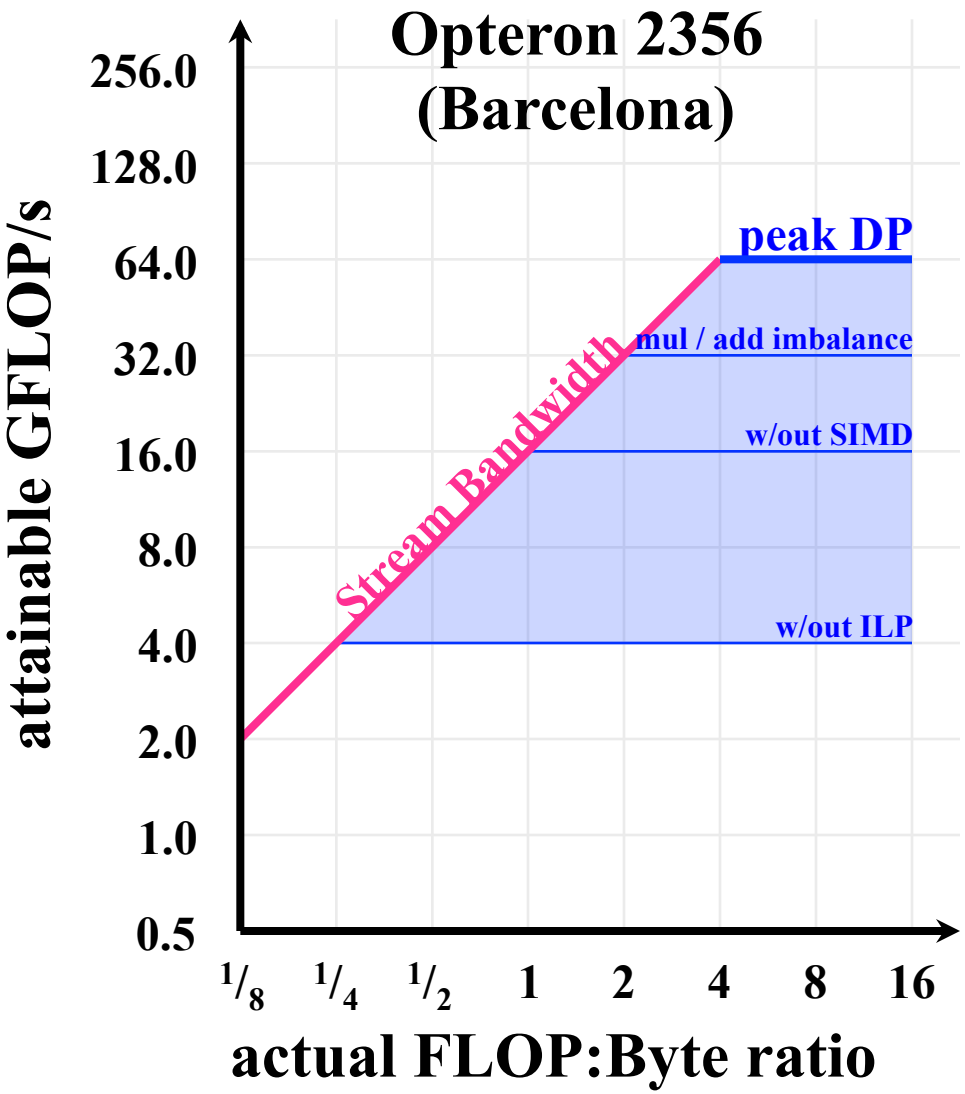
- Opterons have dedicated multipliers and adders.
- If the code is dominated by adds, then attainable performance is half of peak.
- We call these **Ceilings**
- They act like constraints on performance

Roofline Model – Computational Ceilings (2)



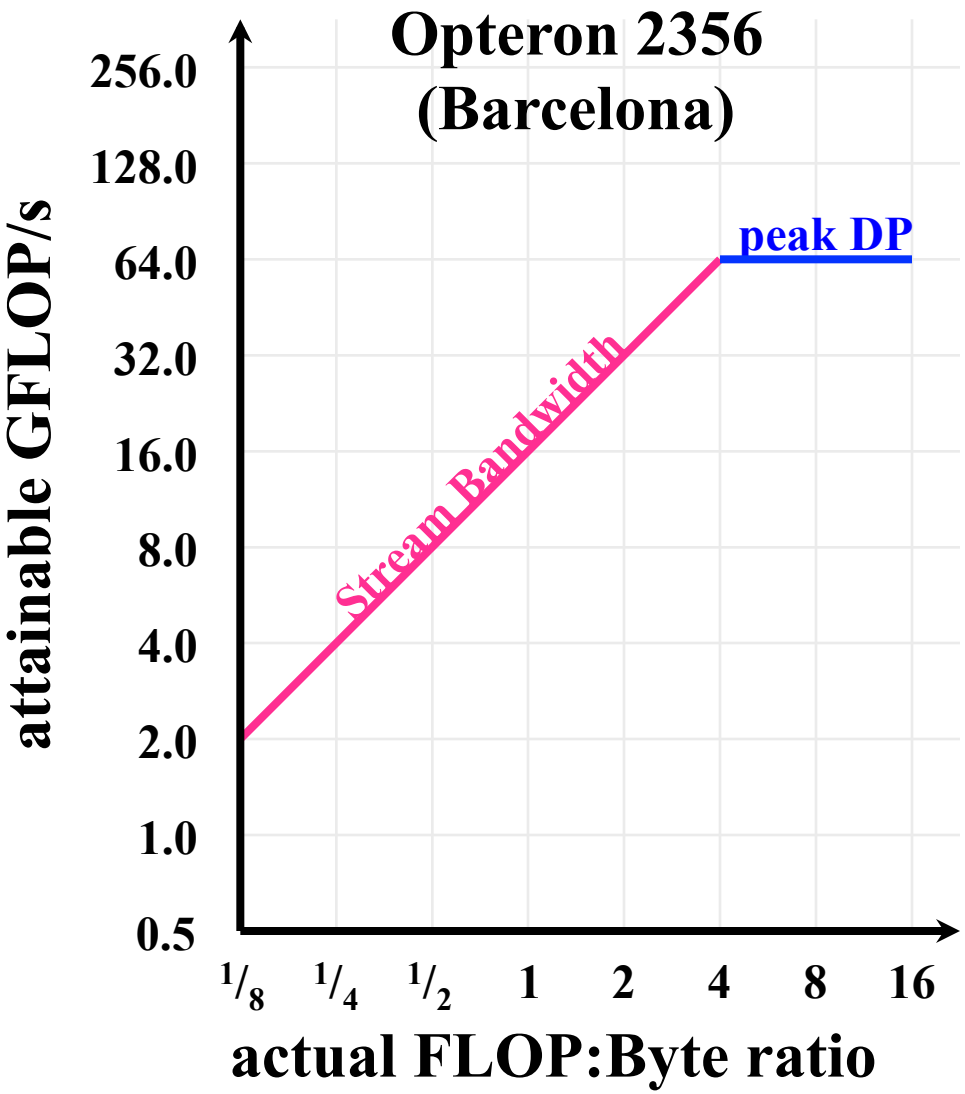
- Opterons have 128-bit datapaths.
- If instructions aren't SIMDized, attainable performance will be halved

Roofline Model – Computational Ceilings (3)



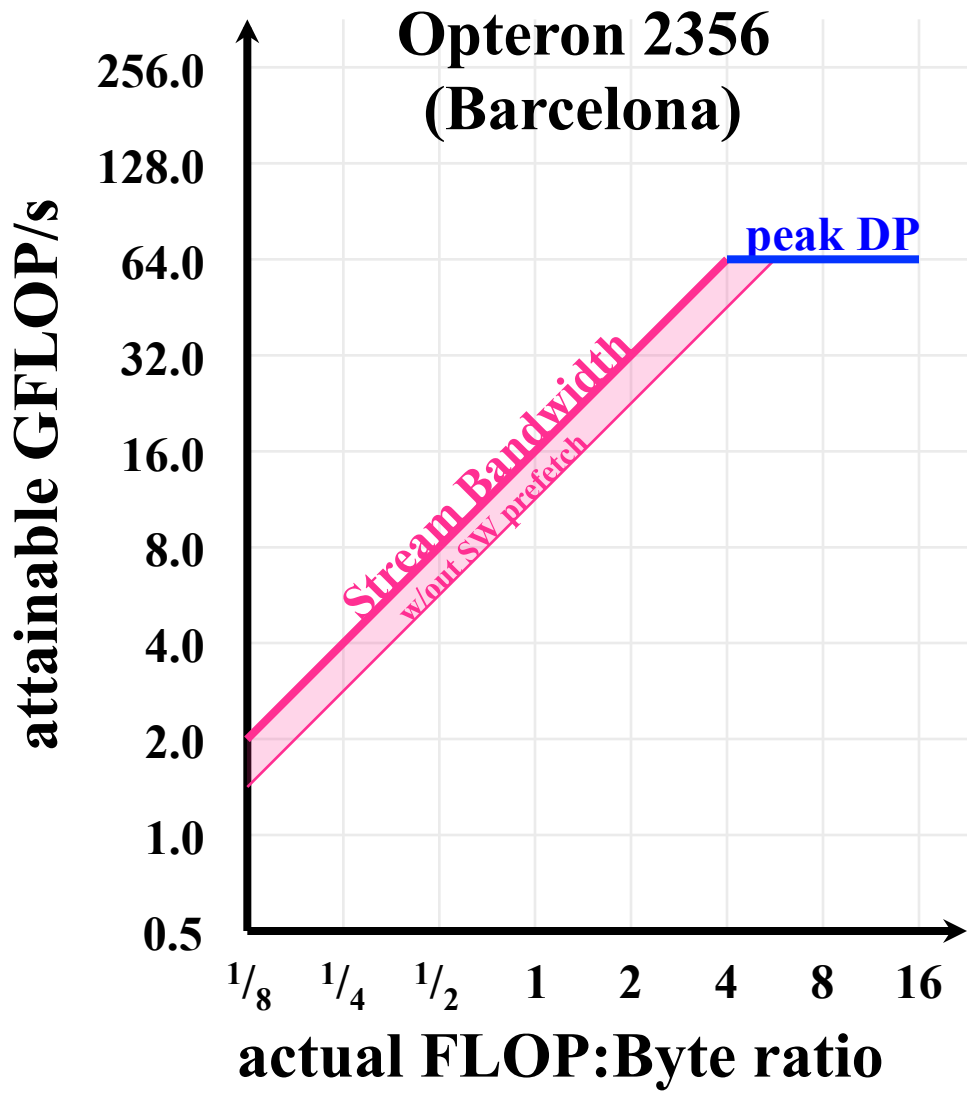
- On Opterons, floating-point instructions have a 4 cycle latency.
- If we don't express 4-way ILP, performance will drop by as much as 4x

Roofline Model – Communication Ceilings



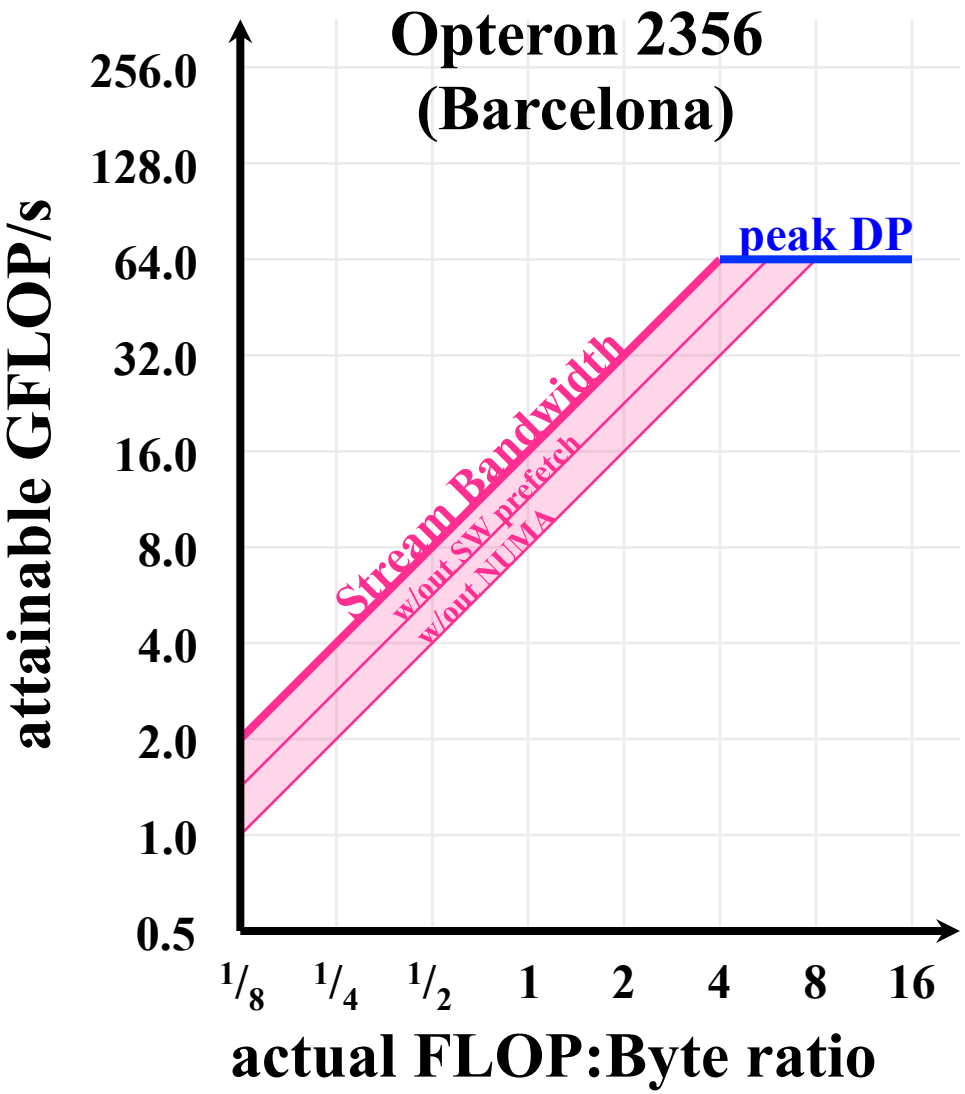
- We can perform a similar exercise taking away parallelism from the memory subsystem

Roofline Model – Communication Ceilings (2)



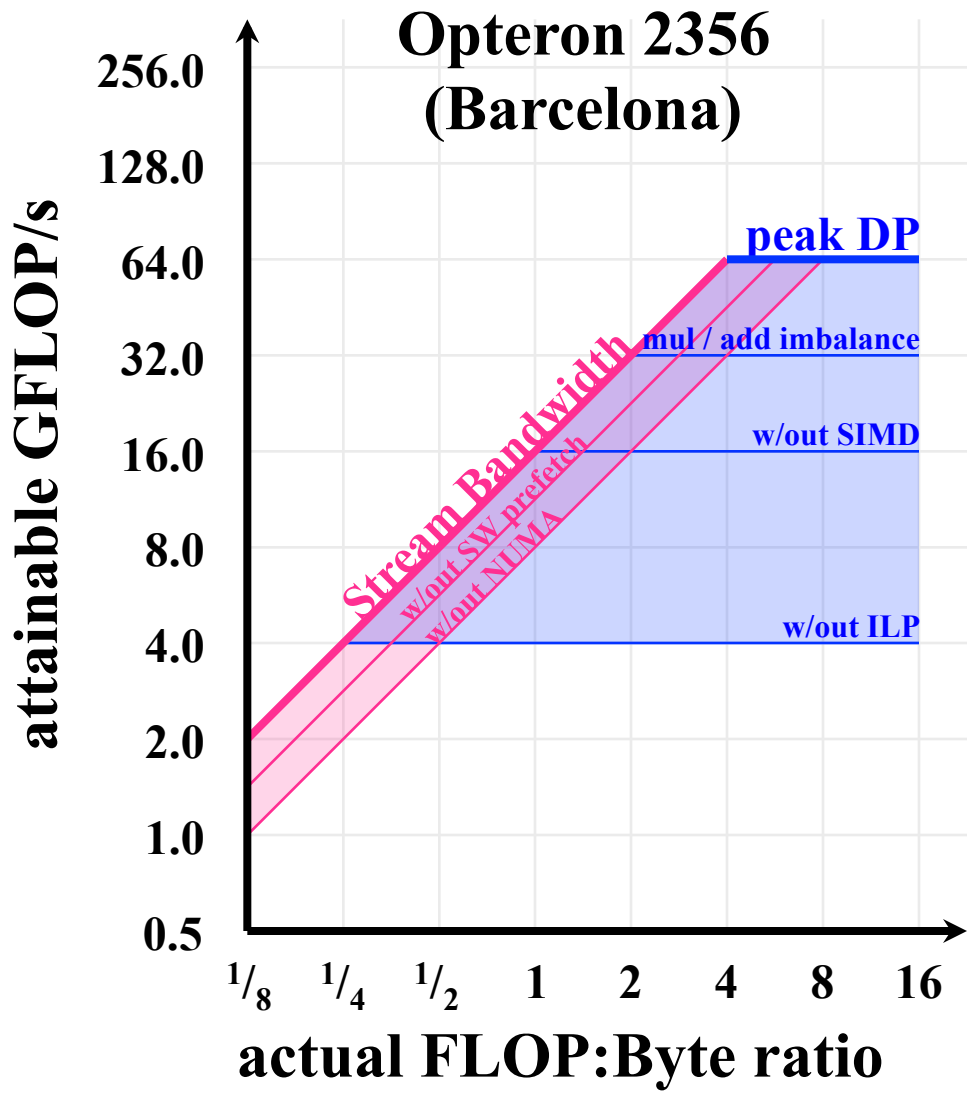
- Explicit software prefetch instructions are required to achieve peak bandwidth

Roofline Model – Communication Ceilings (3)



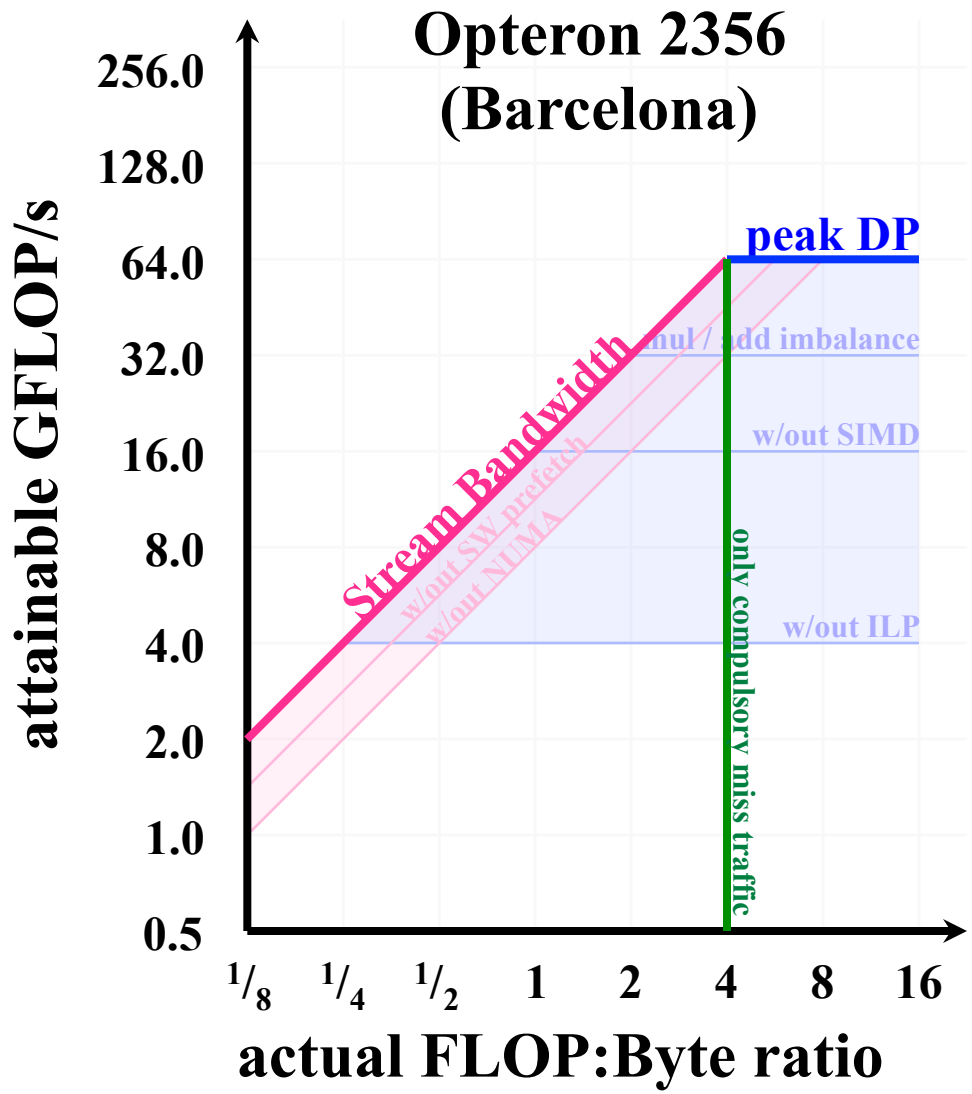
- Opterons are NUMA
- As such memory traffic must be correctly balanced among the two sockets to achieve good Stream bandwidth.
- We could continue this by examining strided or random memory access patterns

Computation + Communication Ceilings



- We may bound performance based on the combination of expressed in-core parallelism and attained bandwidth.

Roofline Model – Locality Walls



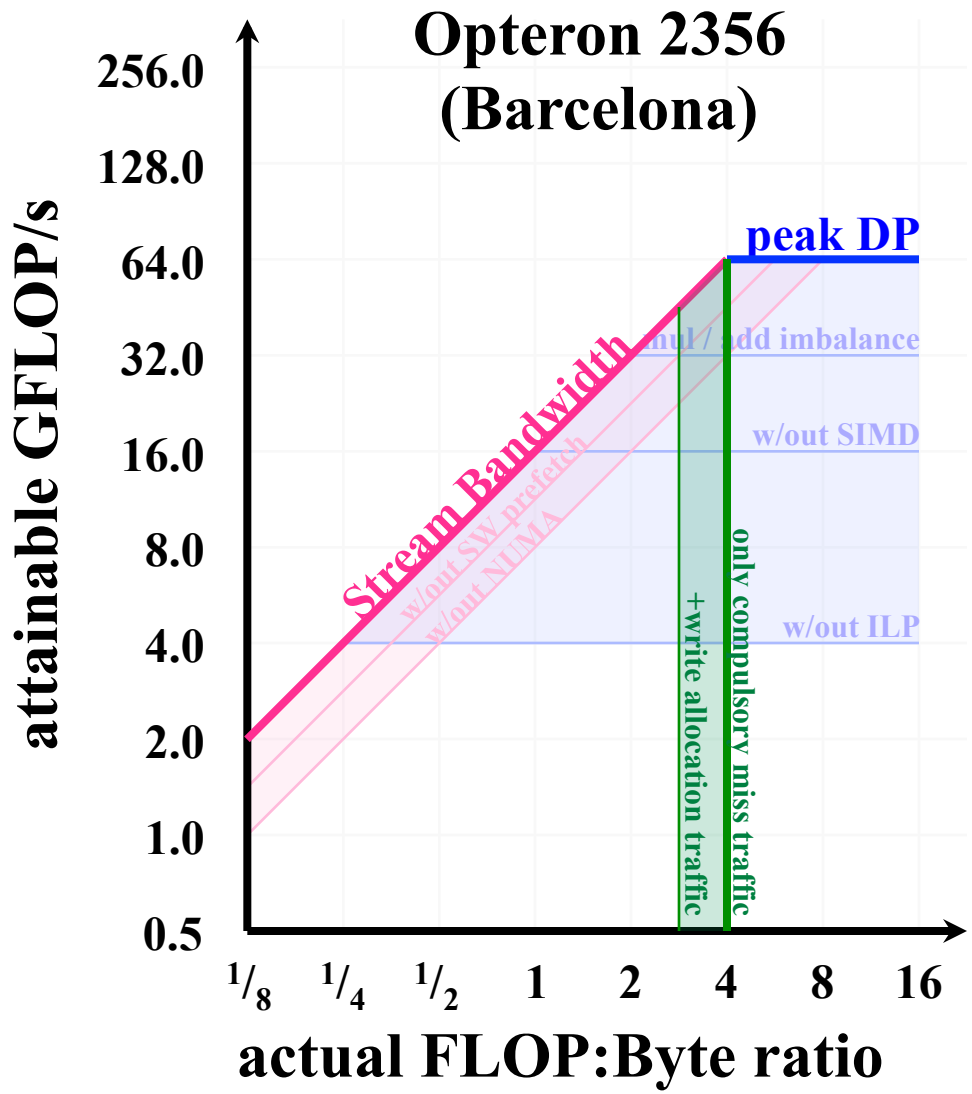
- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

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

Cache Behavior

- Knowledge of the underlying cache operation can be critical
- Caches are organized into lines
  - Lines are organized into sets & ways (associativity)
  - Impacts of conflict, compulsory, and capacity misses are both architecture- and application-dependent
  - **Ultimately they reduce the actual flop:byte ratio.**
- Many caches are write allocate
  - A write allocate cache read in an entire cache line upon a write miss
  - If the application ultimately overwrites that line, the read was superfluous
  - **Further reduces flop:byte ratio**
- Because programs access data in words, but hardware transfers it in 64 or 128B cache lines, spatial locality is key
  - **Array-of-structure data layouts can lead to dramatically lower flop:byte ratios.**

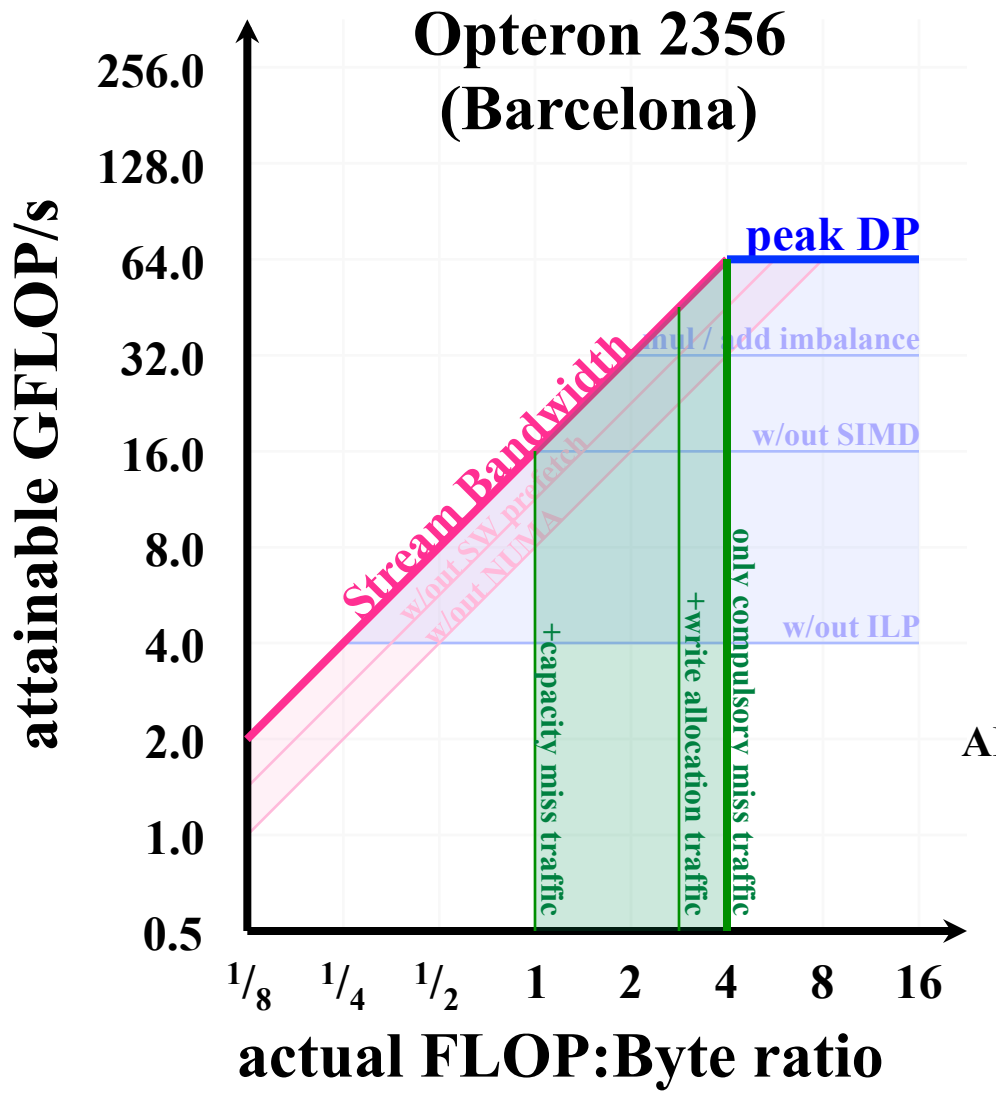
Roofline Model – Locality Walls (2)



- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Allocations + Compulsory Misses}}$$

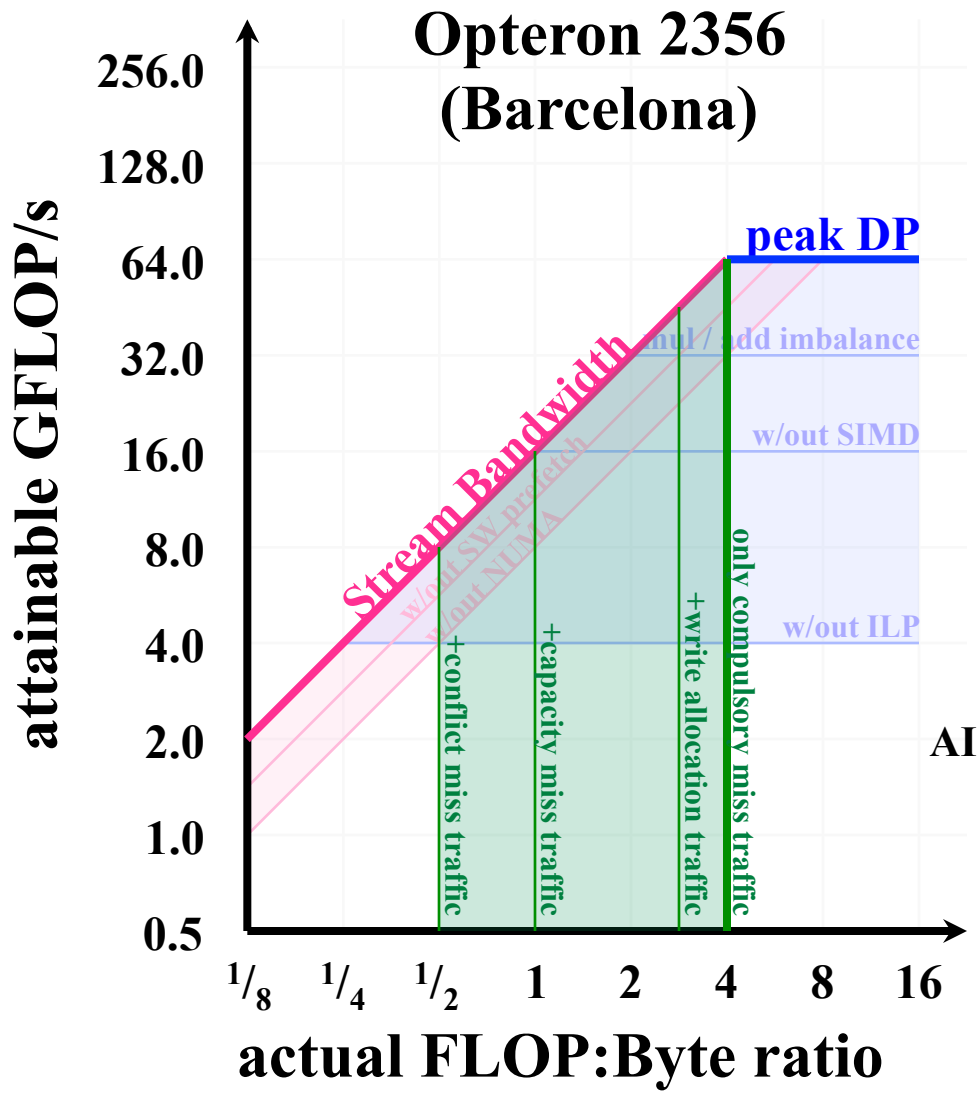
Roofline Model – Locality Walls (3)



- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Capacity} + \text{Allocations} + \text{Compulsory}}$$

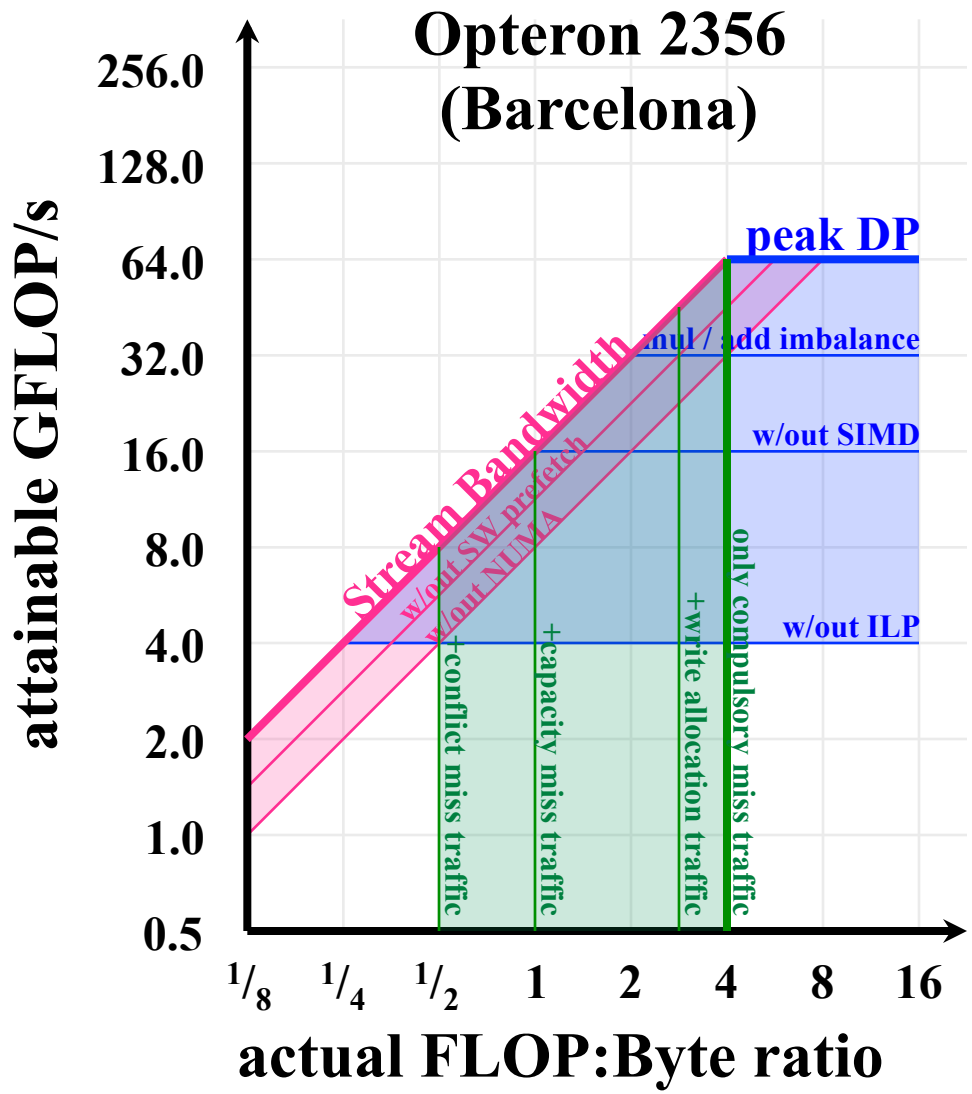
Roofline Model – Locality Walls (4)



- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Conflict} + \text{Capacity} + \text{Allocations} + \text{Compulsory}}$$

Roofline Model – Locality Walls (5)



- Optimizations remove these walls and ceilings **which act to constrain performance.**

Optimization Categorization

Maximizing *(attained)* In-core Performance	**Maximizing *(attained)* Memory Bandwidth**	**Minimizing *(total)* Memory Traffic**

Optimization Categorization

Maximizing In-core Performance

- **Exploit in-core parallelism (ILP, DLP, etc…)**
- **Good (enough) floating-point balance**

Maximizing Memory Bandwidth

Minimizing Memory Traffic

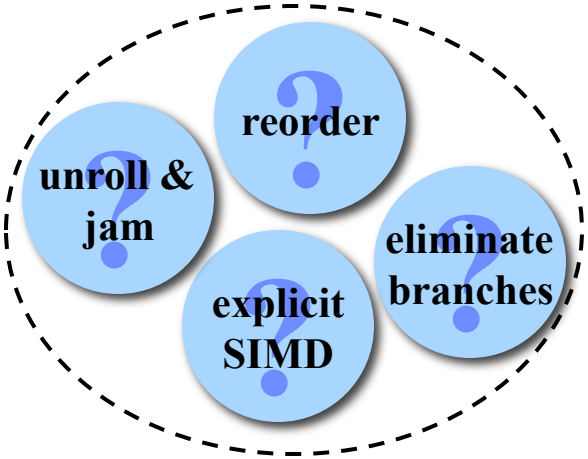
Optimization Categorization

Maximizing In-core Performance

Maximizing Memory Bandwidth

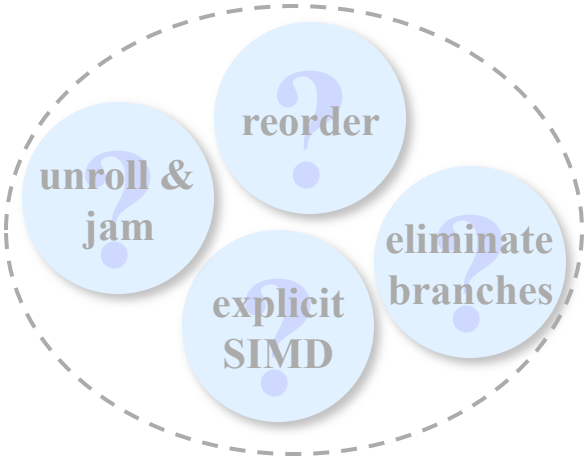
Minimizing Memory Traffic

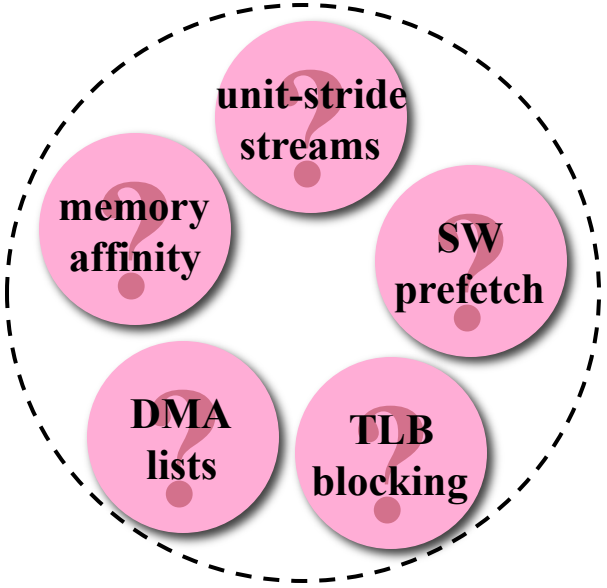
- **Exploit in-core parallelism (ILP, DLP, etc…)**
- **Good (enough) floating-point balance**



Optimization Categorization

Maximizing In-core Performance	**Maximizing Memory Bandwidth**	Minimizing Memory Traffic
•Exploit in-core parallelism (ILP, DLP, etc…)	•Exploit NUMA	
•Good (enough) floating-point balance	•Hide memory latency	
	•Satisfy Little's Law	





Optimization Categorization

Maximizing In-core Performance

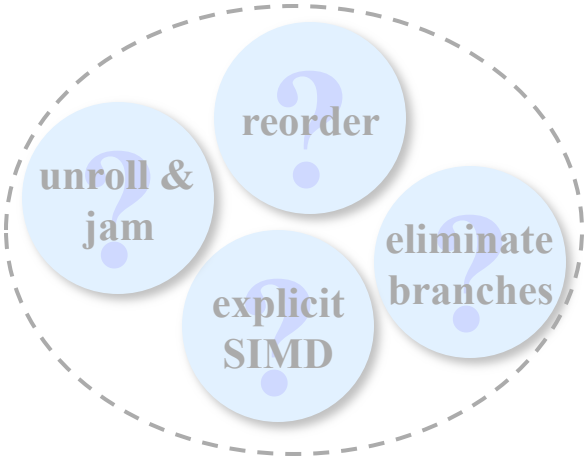
- Exploit in-core parallelism (ILP, DLP, etc…)
- Good (enough) floating-point balance

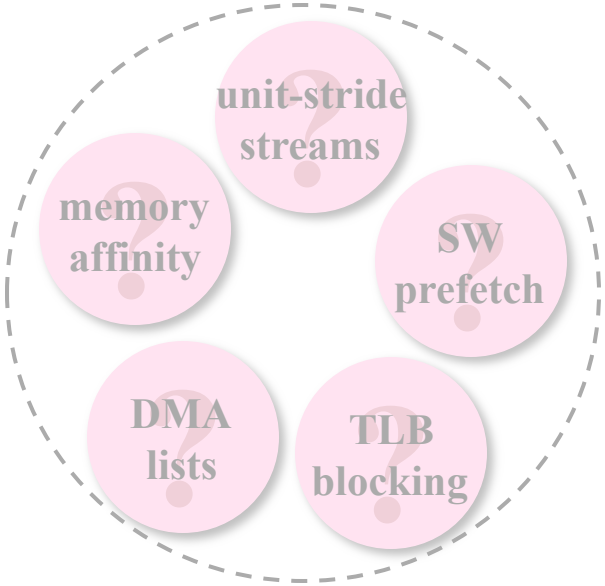
Maximizing Memory Bandwidth

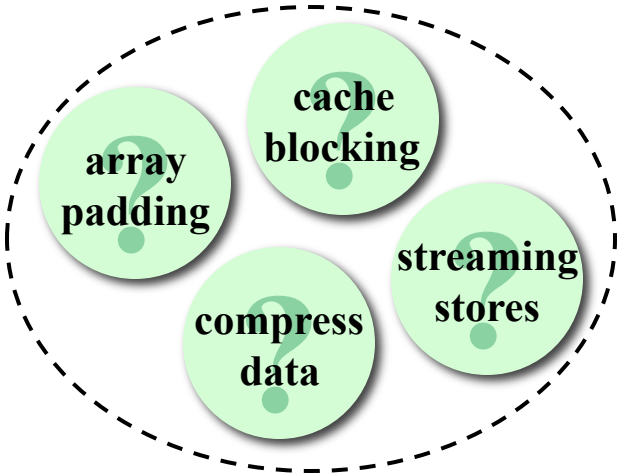
- Exploit NUMA
- Hide memory latency
- Satisfy Little's Law

Minimizing Memory Traffic

Eliminate:
- Capacity misses
- Conflict misses
- Compulsory misses
- Write allocate behavior



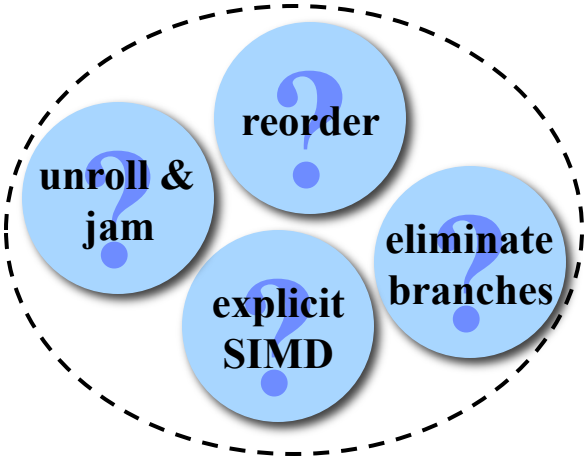




Optimization Categorization

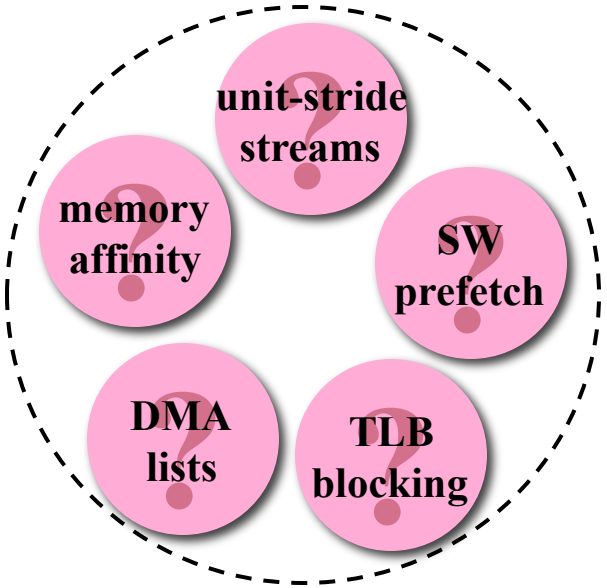
Maximizing In-core Performance

- **Exploit in-core parallelism (ILP, DLP, etc…)**
- **Good (enough) floating-point balance**



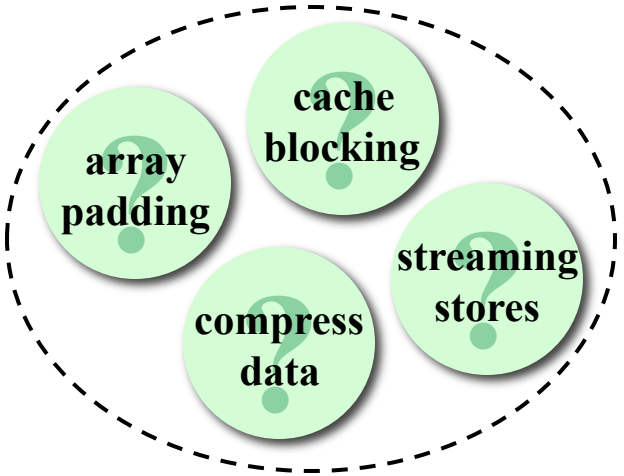
Maximizing Memory Bandwidth

- **Exploit NUMA**
- **Hide memory latency**
- **Satisfy Little's Law**



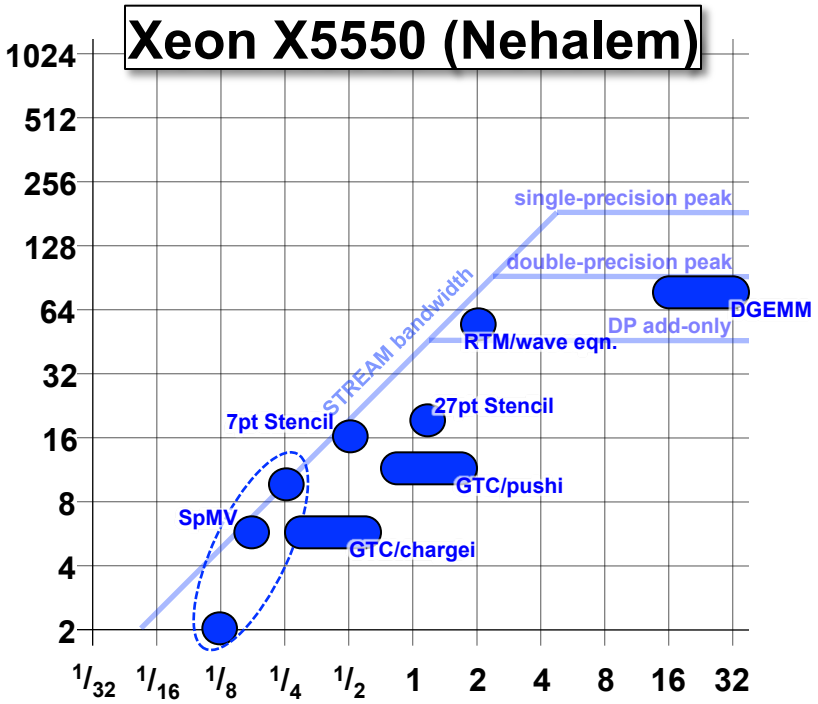
Minimizing Memory Traffic

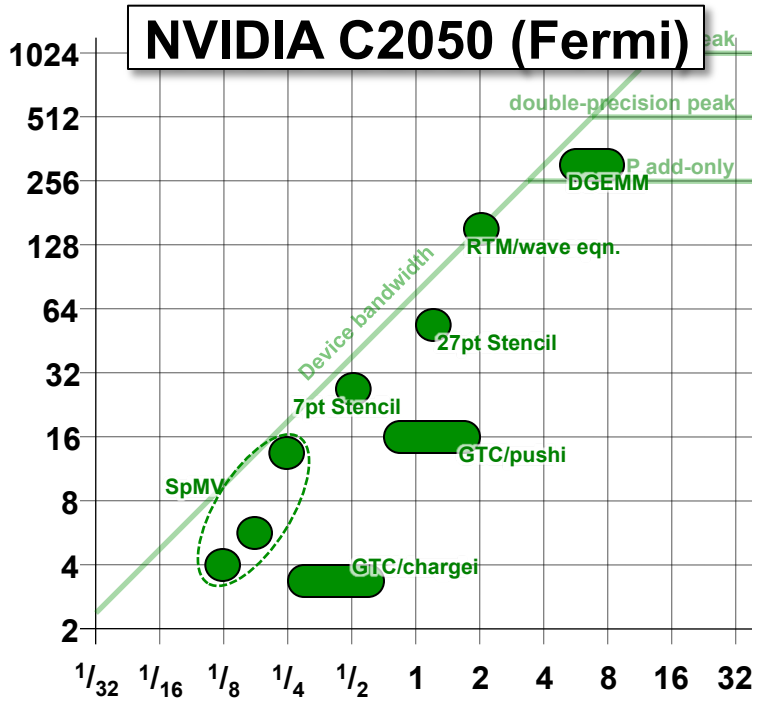
Eliminate:
- **Capacity misses**
- **Conflict misses**
- **Compulsory misses**
- **Write allocate behavior**



Various Kernels

- Williams has examined and heavily optimized a number of kernels and applications for both CPUs and GPUs
- He observe that for most, performance is highly correlated with DRAM bandwidth – particularly on the GPU
- Note, GTC has a strong scatter/gather component that skews STREAM-based rooflines.





Next Class

- Parallel performance analysis
- Parallel performance tools