CESE4130: Computer Engineering

2024-2025, lecture 11

Scaling Out

Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

TU Delft

Delft University of Technology

Announcement

- There will be example questions
- However, do not count on getting similar questions with different numbers

Course objectives

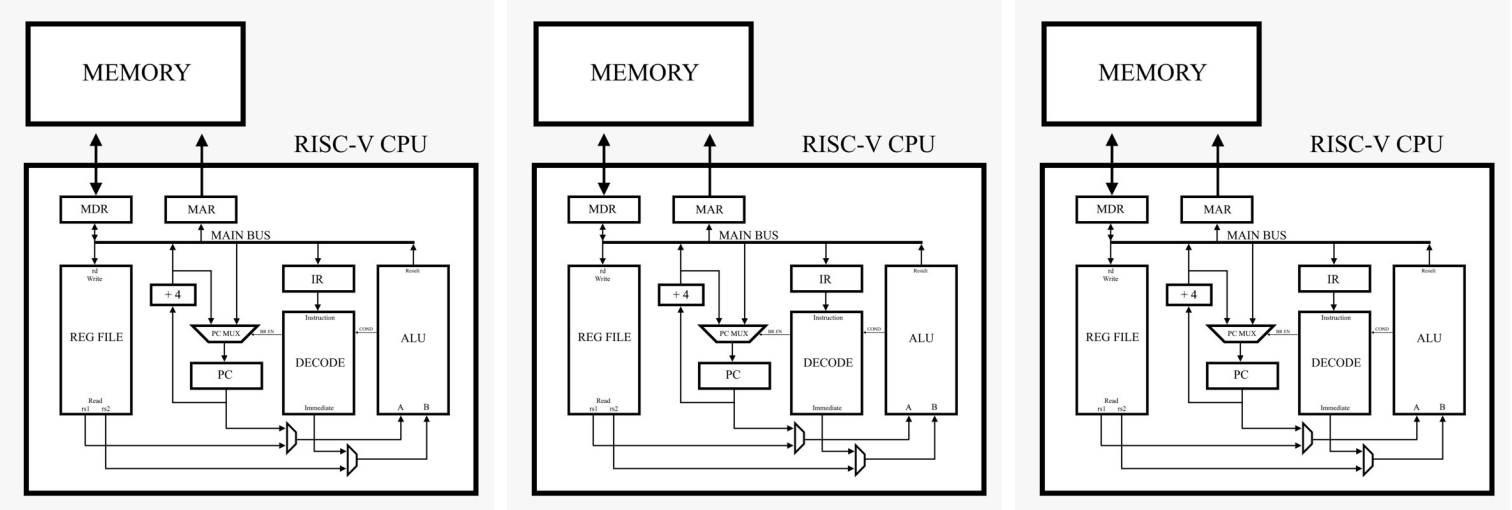
- Describe number representation systems and inter-conversion.
- Perform binary arithmetic operation such as addition and multiplication.
- Explain basic concepts of computer architecture.
- Use logic gates to implement simple combinational circuits.
- Explain system software and operating systems fundamentals, task management, synchronization, compilation, and interpretation.
- Use design and automation tools to perform synthesis and optimization.

Objectives

- Understand the Amdahl and Gustaffson laws
- Explain system scalability
- Get the basics of a widely used performance evaluation model
- More laws

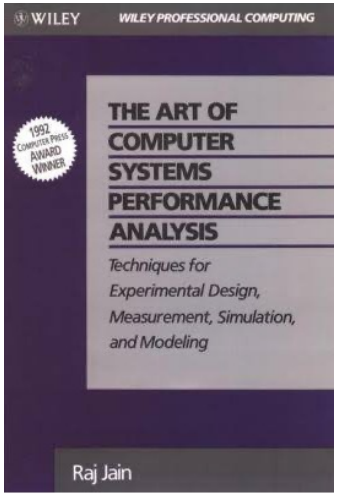
Recap

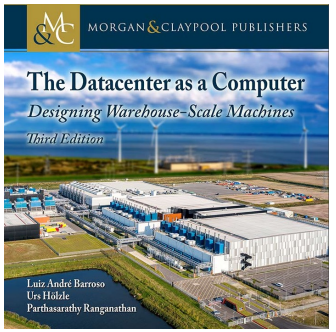
- Understood parallel machines and their interconnect network basics
- Some CUDA intro (to be used in Lab3)
- More(?)
  - our main goal is "**<u>to remove magic</u>**" as you remember



Overview

- The lecture material is collected from various sources
  - Mikko Lipasti (UW-Madison), Samuel Williams (Lawrence Berkeley National Lab) and more
- About Performance Modeling, please refer to Raj Jain
  - https://www.wiley.com/en-us/The+Art+of+Computer+Systems+Performance+Analysis%3A+Techniques+for+Experimental+Design%2C+Measurement%2C+Simulation%2C+and+Modeling-p-9780471503361
- For datacenters, see Luiz Barroso's book …





Speedup and Scalability

- Speedup, Scalability, strong scaling, weak scaling
- Iron law, MIPS, MFLOPS and benchmarks
- Amdahl's law
- Gustafson's law
- Roofline Model
- Operational laws

Performance expectation

- When using one processor, the sequential program runs for 100 seconds. When we use 10 processors, should the program run for 10 times faster?
  - This works only for *embarrassingly parallel computations* – parallel computations that can be divided into completely independent computations that can be executed simultaneously. There may have no interaction between separate processes; sometime the results need to be collected.
    - Embarrassingly parallel applications are the kind that can scale up to a very large number of processors. Examples: Monte Carlo analysis, numerical integration, 3D graphics rendering, etc.
  - In other types of applications, the computation components interact and have dependencies, which prevents the applications from using a large number of processors.

Performance vs Cost

- Which of the following airplanes has the best performance?

Airplane	Passengers	Range (mi)	Speed (mph)
Boeing 737-100	101	630	598
Boeing 747	470	4,150	610
BAC/Sud Concorde	132	4,000	1,350
Douglas DC-8-50	146	8,720	544

- How much faster is the Concorde vs. the 747
- How much bigger is the 747 vs. DC-8?

Performance vs Cost

- Which computer is fastest?

- Not so simple!
  - Scientific simulation – FP performance
  - Program development – Integer performance
  - Database workload – Memory, I/O

Performance of Computers

- Want to buy the fastest computer for what you want to do?
  - Workload is most important
  - Correct measurement and analysis
- Want to design the fastest computer for what the customer wants to pay?
  - Cost is an important criterion

Defining Performance

- What is important to whom?
- Computer system user
  - Minimize elapsed time for program = *time_end – time_start*
  - Called response time (aka **latency**)
- Computer center manager
  - Maximize completion rate = *#jobs/second*
  - Called **throughput**

Response Time vs. Throughput

- Is throughput = 1/av. response time?
  - Only if NO **overlap**
  - Otherwise, throughput > 1/av. response time
  - E.g., a lunch buffet – assume 5 entrees
  - Each person takes 2 minutes/entrée
  - Throughput is 1 person every 2 minutes
  - BUT time to fill up tray is 10 minutes
  - Why and what would the throughput be otherwise?
  - 5 people simultaneously filling tray (overlap)
  - Without overlap, throughput = 1/10

What is Performance for Computer Architects?

- Computer architects' view
  - CPU time = time spent running a program
- Intuitively, bigger should be faster, so:
  - Performance = 1/X time, where X is response, CPU execution, etc.
- Elapsed time = CPU time + I/O wait time
- Let's consider only CPU time (*any comments?*)

Improve Performance

- Improve (a) response time or (b) throughput?
  - Faster CPU
  - Helps both (a) and (b)
- Add more CPUs
  - Helps (b) and perhaps (a) due to less queueing

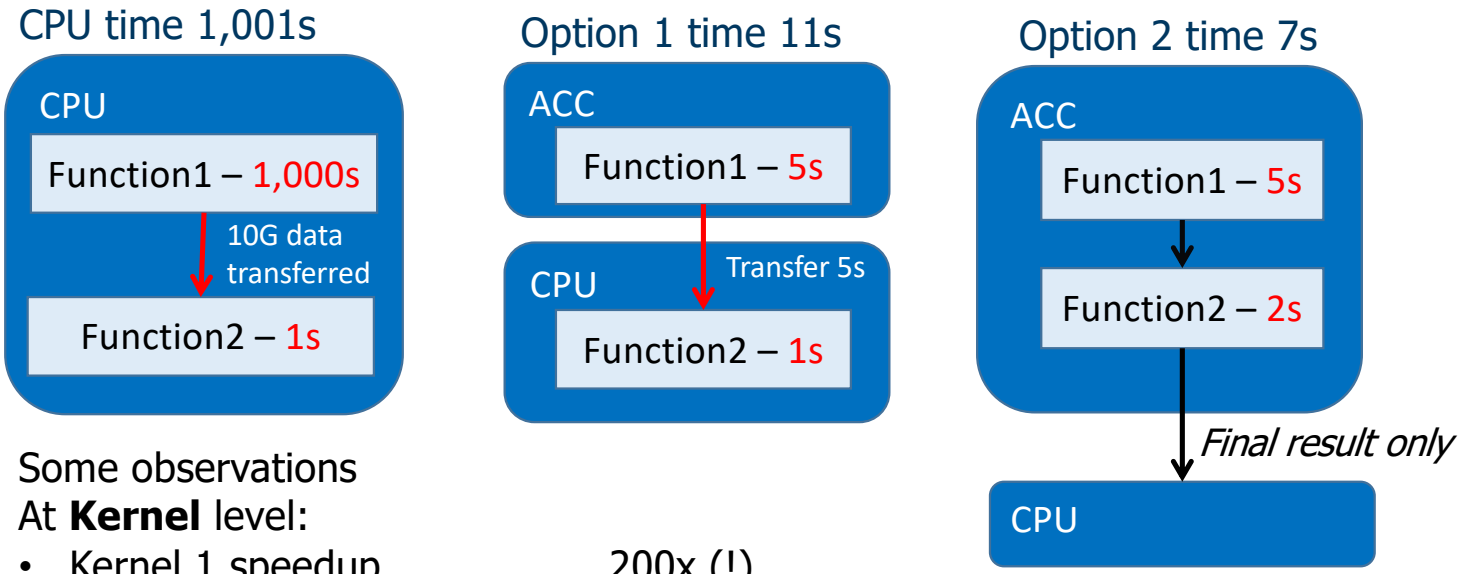
Performance Comparison

- Machine A is n times faster than machine B

  iff *perf(A)/perf(B) = time(B)/time(A) = n*
- Machine A is x% faster than machine B

  iff *perf(A)/perf(B) = time(B)/time(A) = 1 + x/100*
- e.g., time(A) = 10s, time(B) = 15s
  - 15/10 = 1.5 => A is 1.5 times faster than B
  - 15/10 = 1.5 => A is 50% faster than B

What about "decelerating" accelerators?



Some observations

At **Kernel** level:

- Kernel 1 speedup 200x (!)
- Kernel 2 "speedup" 0.5x (!)

At **System** level:

- Option 1 (Kernel 1 only) speedup **91x**
- Option 2 (Kernels 1 and 2) speedup **143x**

Some thinking about computation and data movements is required

Breaking Down Performance

- A program is broken into instructions
  - Hardware (u-architecture) is aware of instructions, not programs
- At lower level, HW breaks instructions into cycles
  - Lower level state machines change state every cycle
- For example:
  - 1GHz Snapdragon runs 1,000M cycles/sec, 1 cycle = 1ns
  - 2.5GHz Core i7 runs 2.5G cycles/sec, 1 cycle = 0.25ns

Iron Law

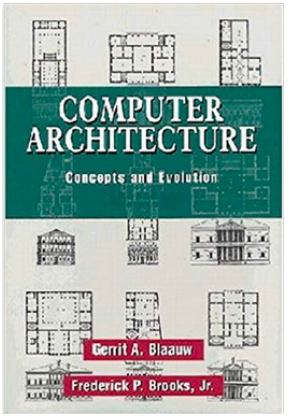
$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

Architecture --> Implementation --> Realization

Compiler Designer **Processor Designer** **Chip Designer**



Remember Blaauw Brooks?

Iron Law

- Instructions/Program
  - Instructions executed, not static code size
  - Determined by algorithm, compiler, ISA
- Cycles/Instruction
  - Determined by ISA and CPU organization
  - Overlap among instructions reduces this term
- Time/cycle
  - Determined by technology, organization, clever circuit design

The overall CPU architect goal

- Minimize time which is the product, NOT isolated terms
- Common error to miss terms while devising optimizations
  - e.g., ISA change to decrease instruction count
  - BUT leads to CPU organization which makes clock slower

- Bottom line: terms are inter-related

Other Metrics

- MIPS and MFLOPS
- MIPS = instruction count/(execution time x 10^6)

  = clock rate/(CPI x 10^6)
- But MIPS has **serious** shortcomings

Problems with MIPS

- **Example:** without FP hardware, an FP op may take 50 single-cycle instructions
- With FP hardware, only one 2-cycle instruction

- Thus, adding FP hardware:
  - CPI increases (why?)
  - Instructions/program decreases (why?)
  - Total execution time decreases

50/50 => 2/1
50 => 1
50 => 2

50 MIPS => 2 MIPS

- BUT, MIPS gets worse!

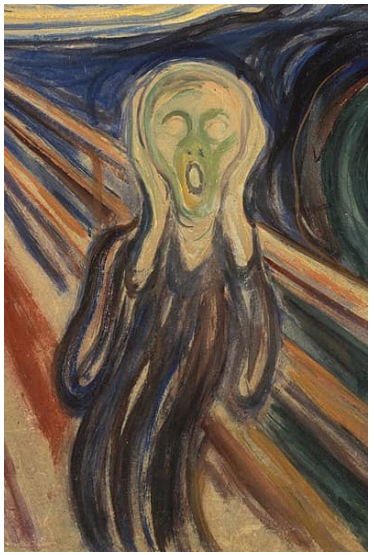
Problems with MIPS

- **Ignores program**
- Usually used to quote peak performance
  - Ideal conditions => guaranteed not to exceed!

- When is MIPS ok?
  - Same compiler, same ISA
  - e.g., same binary running on AMD Phenom, Intel Core i7
  - Why? Instructions/program is constant and can be ignored

Other Metrics

- MFLOPS = FP ops in program/(execution time x 106)
- Assuming FP ops independent of compiler and ISA
  - Often safe for numeric codes: matrix size determines # of FP ops/program
  - However, not always safe:
    - Missing instructions (e.g., FP divide)
    - Optimizing compilers
- Relative MIPS and normalized MFLOPS
  - Just adds to the confusion

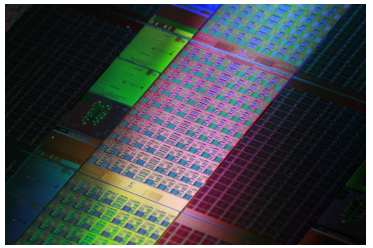


Which Programs?

- Execution time of what program?
- Best case – your always run the same set of programs
  - Port them and time the whole workload
- In reality, use benchmarks
  - Programs chosen to measure performance
  - Predict performance of actual workload
  - Saves effort and money
  - Representative? Honest? Benchmarketing…

But also: Synthetic Programs, Kernels, Instruction Mixes, Addition/MUL Instruction (?)

Intel Polaris (2006)
80 cores @ 4GHz
dual- SPFP MACs /cycle
1TFLOP < 100W



How to Average?

	Machine A	Machine B
Program 1	1	10
Program 2	1,000	100
Total	1,001	110

- One answer: for total execution time, how much faster is B? 9.1x

How to Average?

- Another: Arithmetic Mean (AM) (same result)
- Arithmetic mean of times:
- AM(A) = 1001/2 = 500.5
- AM(B) = 110/2 = 55
- 500.5/55 = 9.1x
- Valid only if programs run equally often, so use weighted arithmetic mean:

$$\left\{\sum_{i=1}^{n} time(i)\right\} \times \frac{1}{n}$$

$$\left\{\sum_{i=1}^{n} \left(weight(i) \times time(i)\right)\right\} \times \frac{1}{n}$$

More in Chapter 12 of Raj Jains' book

Other Averages

- e.g., 30 km/h for first 10 km, then 90 km/h for next 10 km, what is average speed?
- Average speed = (30+90)/2 **WRONG!**
- Average speed = total distance / total time

  = (20 / (10/30 + 10/90))

  = **45 km/h**

Harmonic Mean

- Harmonic mean of rates = $$\frac{n}{\left\{\sum_{i=1}^{n} \frac{1}{rate(n)}\right\}}$$
- Use HM if forced to start and end with rates (e.g., reporting MIPS or MFLOPS)
- Why?
  - Rate has time in denominator
  - Mean should be proportional to inverse of sums of time (not sum of inverses)
  - See: J.E. Smith, "Characterizing computer performance with a single number," CACM Volume 31 , Issue 10 (October 1988), pp. 1202-1206.

Dealing with Ratios

	Machine A	Machine B
Program 1	1	10
Program 2	1,000	100
Total	1,001	110

- If we take ratios with respect to machine A

	Machine A	Machine B
Program 1	1	10
Program 2	1	0.1

Dealing with Ratios

- Average for machine A is 1, average for machine B is 5.05
- If we take ratios with respect to machine B

	Machine A	Machine B
Program 1	0.1	1
Program 2	10	1
Average	5.05	1

- Can’t both be true!!!
- Don’t use arithmetic mean on ratios!

Geometric Mean

- Use geometric mean for ratios
- Geometric mean of ratios = $\sqrt[n]{\prod_{i=1}^{n} ratio(i)}$

- Independent of reference machine
- In the example, GM for machine a is 1, for machine B is also 1
  - Normalized with respect to either machine

But …

- GM of ratios is not proportional to total time
- AM in example says machine B is 9.1 times faster
- GM says they are equal
- If we took total execution time, A and B are equal only if
  - Program 1 is run 100 times more often than program 2
- Generally, GM will mispredict for three or more machines

Summary on averaging

- Use AM for times
- Use HM if forced to use rates
- Use GM if forced to use ratios

- Best of all, use unnormalized numbers to compute time

See also Chapter 12 of Raj Jains' book, especially Section 12.4

Benchmarks: Standard Performance Evaluation Corporation

- System Performance Evaluation Cooperative (SPEC)
  - Formed in 80s to combat benchmarketing
  - SPEC89, SPEC92, SPEC95, SPEC2000, SPEC CPU2006, 2017
- SPEC CPU® 2017: 43 benchmarks in four suites:
  - SPECspeed® 2017 Integer
  - SPECspeed® 2017 Floating Point
  - SPECrate® 2017 Integer
  - SPECrate® 2017 Floating Point
  - optional metric for energy consumption.

Benchmarks (INT 2017 and 2000)

SPECrate®2017 Integer	SPECspeed®2017 Integer	Language [1]	KLOC [2]	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

Benchmark	Description
164.gzip	Compression
175.vpr	FPGA place and route
176.gcc	C compiler
181.mcf	Combinatorial optimization
186.crafty	Chess
197.parser	Word processing, grammatical analysis
252.eon	Visualization (ray tracing)
253.perlbmk	PERL script execution
254.gap	Group theory interpreter
255.vortex	Object-oriented database
256.bzip2	Compression
300.twolf	Place and route simulator

Benchmarks (FP 2017 and 2000)

SPECrate®2017 Floating Point	SPECspeed®2017 Floating Point	Language [1]	KLOC [2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

[1] For multi-language benchmarks, the first one listed determines library and link options (details)

[2] KLOC = line count (including comments/whitespace) for source files used in a build / 1000

Benchmark	Description
168.wupwise	Physics/Quantum Chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver: 3D potential field
173.applu	Parabolic/elliptic PDE
177.mesa	3-D graphics library
178.galgel	Computational Fluid Dynamics
179.art	Image Recognition/Neural Networks
183.equake	Seismic Wave Propagation Simulation
187.facerec	Image processing: face recognition
188.ammp	Computational chemistry
189.lucas	Number theory/primality testing
191.fma3d	Finite-element Crash Simulation
200.sixtrack	High energy nuclear physics accelerator design
301.apsi	Meteorology: Pollutant distribution

Benchmarks Pitfalls

- Benchmark not representative
  - Your workload is I/O bound, SPEC is useless
- Benchmark is too old
  - Benchmarks age poorly; benchmarketing pressure causes vendors to optimize compiler/hardware/software to benchmarks
  - Need to be periodically refreshed

Scalability

- Scalability of a program measures how many processors that the program can make effective use of.
  - For a computation represented by a computation graph, parallelism is a good indicator of scalability.

Speedup and strong scaling

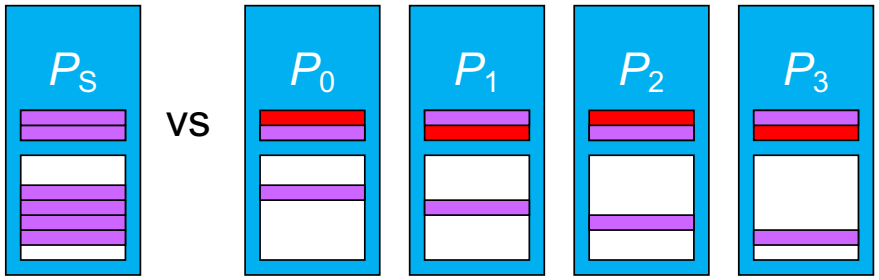
- Let T_1 be the execution time for a computation to run on **one processor** and T_p be the execution time for the computation (with the same input – same problem) to run on **P processors**

$$speedup(P) = \frac{T_1}{T_P}$$

  - Factor by which the use of P processors speeds up execution time relative to one processor for the same problem
  - Since the problem size is fixed, this is referred to as **strong scaling**
  - Given a computation graph, what is the highest speedup that can be achieved?

Speedup

- $speedup(P) = \frac{T_1}{T_P}$
- Typically, $1 \leq speedup(P) \leq P$
- The speedup is ideal if $speedup(P) = P$
- Linear speedup: $speedup(P) = k \times P$ for some constant $0 < k < 1$



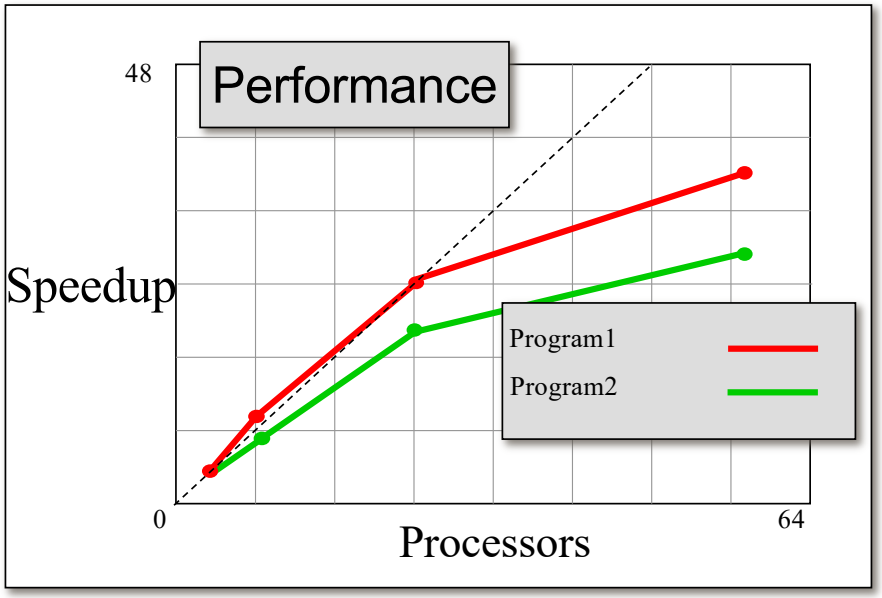
Q: Can speedup be > P

Efficiency

- The efficiency of an || program using P processors is:

$$Efficiency = speedup(P) / P$$

  - Efficiency estimates how well-utilized the processors are in running the parallel program
  - Ideal speedup means Efficiency = 1 (100% efficiency)



Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
  - Need to have computation, communication in same ratio
  - Great sensitivity to the T_S value
    - T_S should be the time of the best sequential program on 1 processor of the ||-machine
    - $T_{P=1} \neq T_S$ Measures **relative speedup**

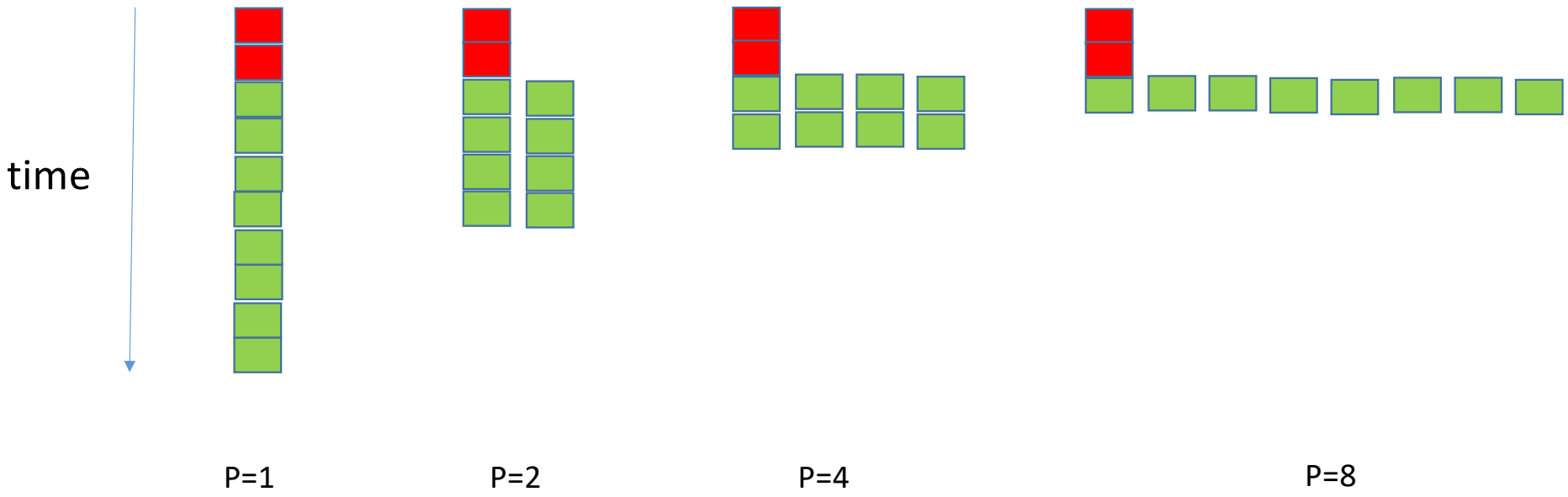
Relative speedup is often important
but it must be labeled as such

Amdahl's Law (fixed size speedup, strong scaling)

- Given a program, let *f* be the fraction that must be sequential and *1-f* be the fraction that can be parallelized
- $T_P = f\ T_1 + \frac{(1-f)T_1}{P}$
- $Speedup(P) = \frac{T_1}{T_P} = \frac{T_1}{f\ T_1 + \frac{(1-f)T_1}{P}} = \frac{1}{f + (1-f)/P}$
- When $P \to \infty, Speedup(P) = \frac{1}{f}$
- Original paper: *Amdahl, Gene M. (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" . AFIPS Conference Proceedings (30): 483–485.*

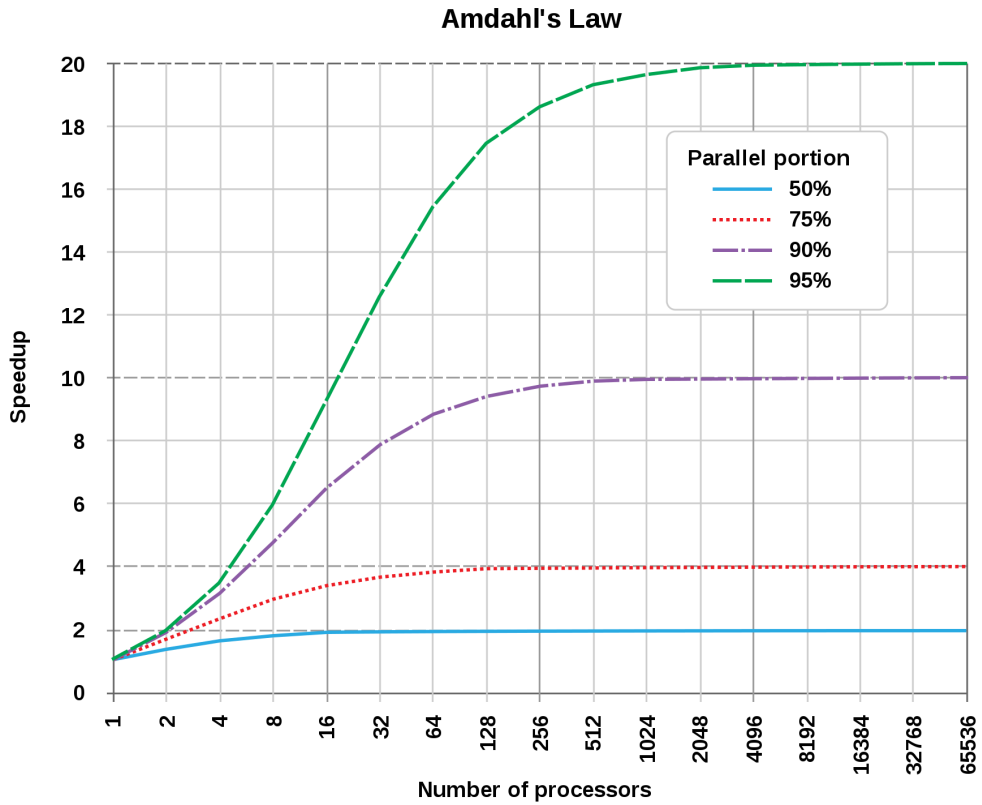
Amdahl's Law

Amdahl's law: As P increases, the percentage of work in the parallel region reduces, performance is more and more dominated by the sequential region



Implications of Amdahl's Law

- For strong scaling, the speedup is bounded by the percentage of sequential portion of the program, **not** by the number of processors!
- Strong scaling will be hard to achieve for many programs

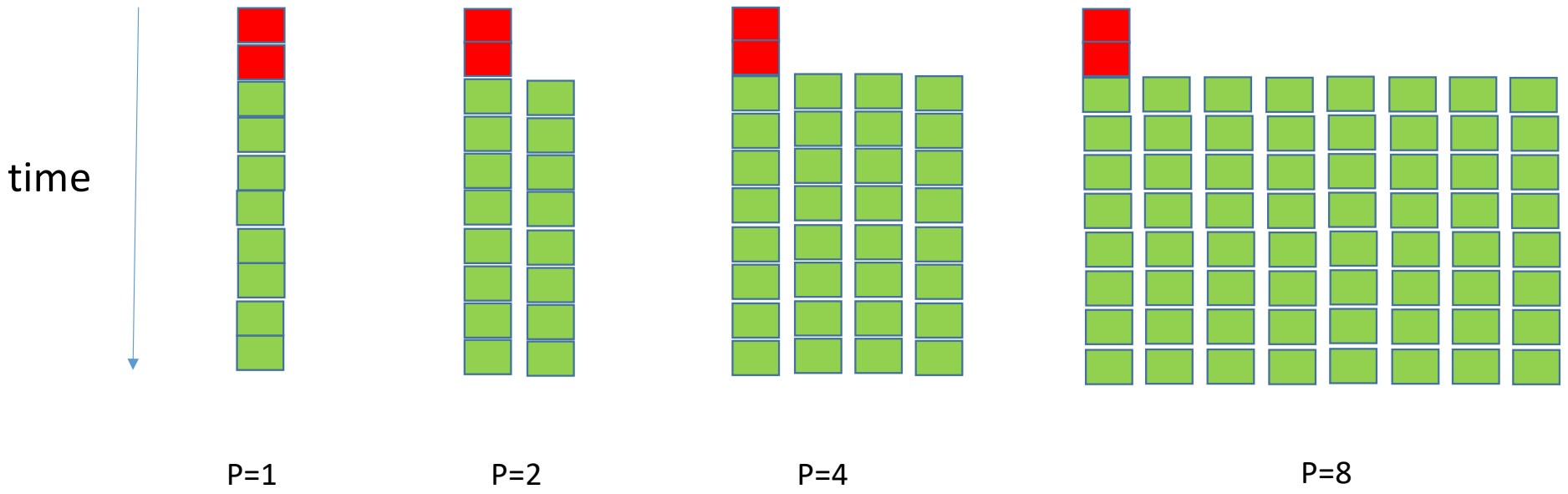


Gustafson's Law (scaled speedup, weak scaling)

- Large scale parallel/distributed systems are expected to allow for solving problem faster or larger problems
  - Amdahl's Law indicates that there is a limit on how faster it can go
  - How about bigger problems? This is what Gustafson's Law sheds lights on!
- In Amdahl's law, as the number of processors increases, the amount of work in each node decreases (*more processors sharing the parallel part*)
- In Gustafson's law, as the number of processors increases, the amount of work in each node remains the same (*doing more work collectively*)

Gustafson's Law

Gustafson's law: As P increases, the total work on each process remains the same. So the total work increases with P.



Gustafson's Law (scaled speedup, weak scaling)

- The work on each processor is 1 (f is the fraction for sequential program, (1-f) is the fraction for parallel program.
- With P processor (with the same $T_P = 1$), the total amount of useful work is $f + (1 - f)P$. Thus, $T_1 = f + (1 - f)P$.
- Thus, $speedup(P) = f + (1 - f)P$.

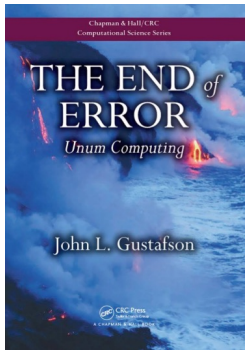
No of PEs	Strong scaling speedup (Amdalh's law, f = 10%)	Weak scaling speedup (Gustafson's law, f = 10%)
2	1.82	1.9
4	3.07	3.7
8	4.71	7.3
16	6.40	14.5
100	9.90	90.1

Implication of Gustafson's Law

- For weak scaling, $speedup(P) = f + (1 - f)P$
  - Speedup is now proportional to P.
- Scalability is much better when the problem size can increase
  - Many application can use more computing power to solve larger problems
    - Weather prediction, large deep learning models.

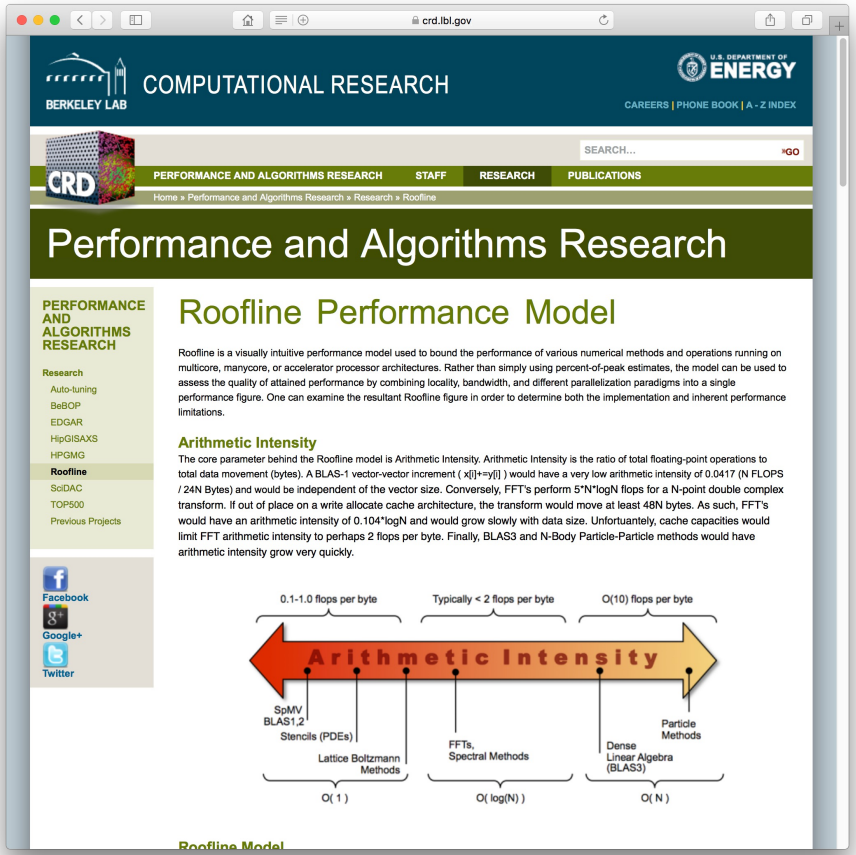
- *Gustafson, John L. (May 1988). "Reevaluating Amdahl's Law". Communications of the ACM. **31** (5): 532–3.*

he also came up with



Modeling Performance: Roofline Model

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

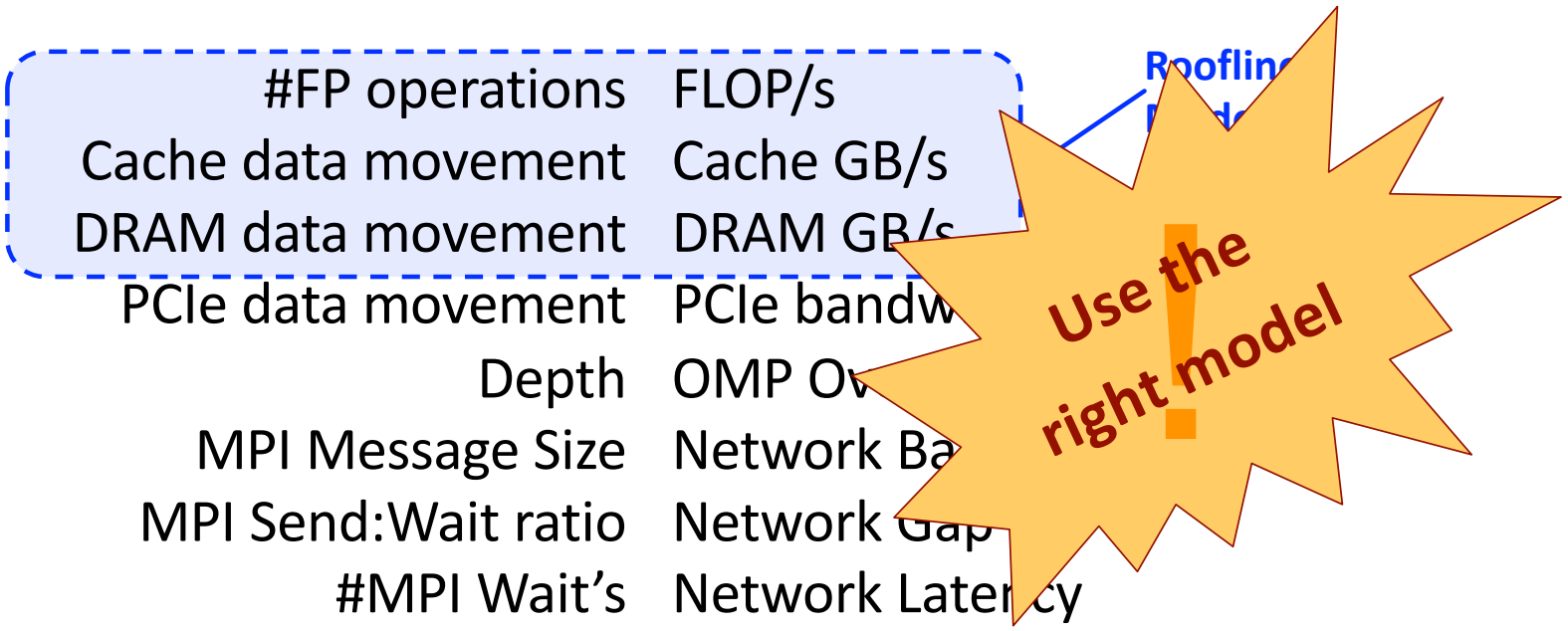


https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

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

Performance Models

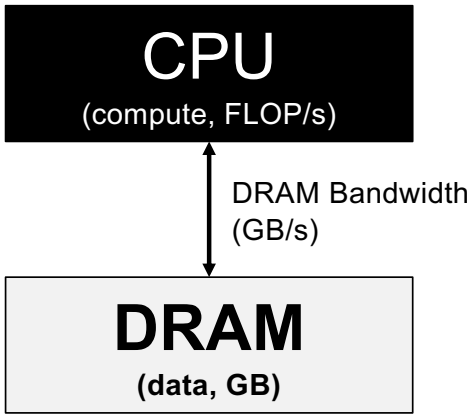
- Because there are so many components, performance models often conceptualize the system as being dominated by one or more of these components.



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

DRAM Roofline

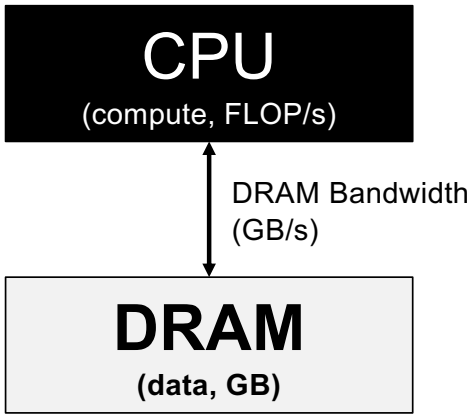
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

DRAM Roofline

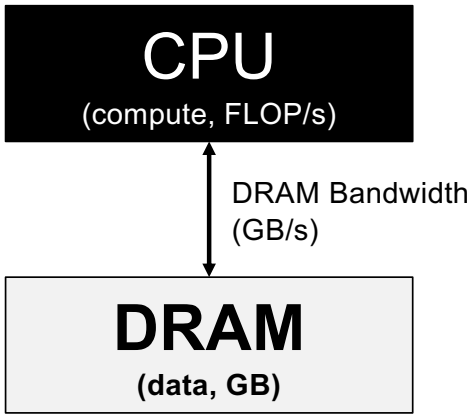
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

DRAM Roofline

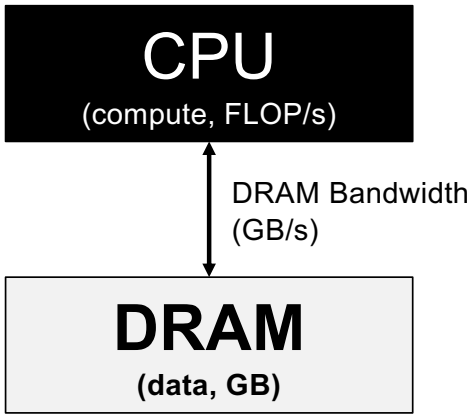
- One could hope to always attain peak performance (FLOP/s)
- However, finite locality (reuse) and bandwidth limit performance.
- Assume:
  - Idealized processor/caches
  - Cold start (data in DRAM)



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

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)



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

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

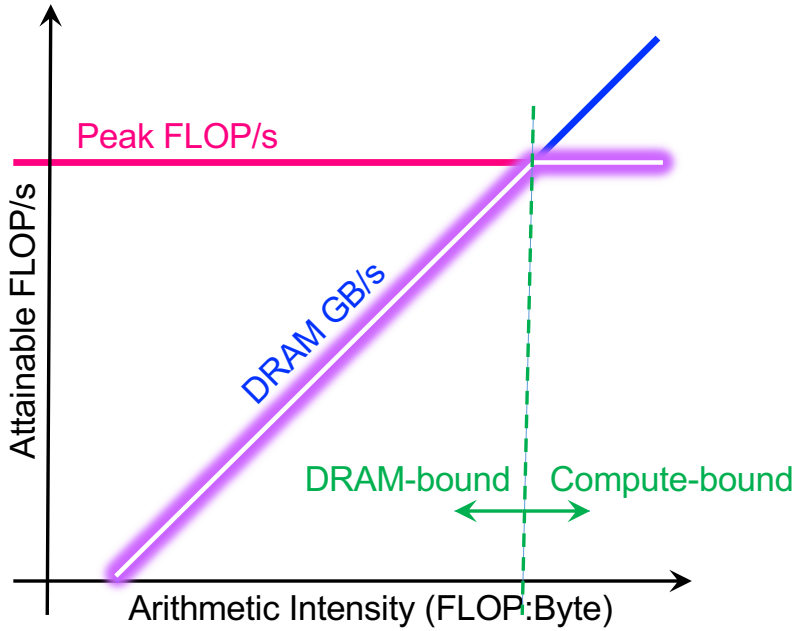
Arithmetic Intensity

- The most important concept in Roofline is **Arithmetic Intensity**

- Measure of data locality (data reuse)
- Ratio of **Total Flops** performed to **Total Bytes** moved
- For the DRAM Roofline…
  - Total Bytes to/from DRAM and includes all cache and prefetcher effects
  - Can be very different from total loads/stores (bytes requested)
  - Equal to ratio of sustained GFLOP/s to sustained GB/s (time cancels)

DRAM Roofline

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



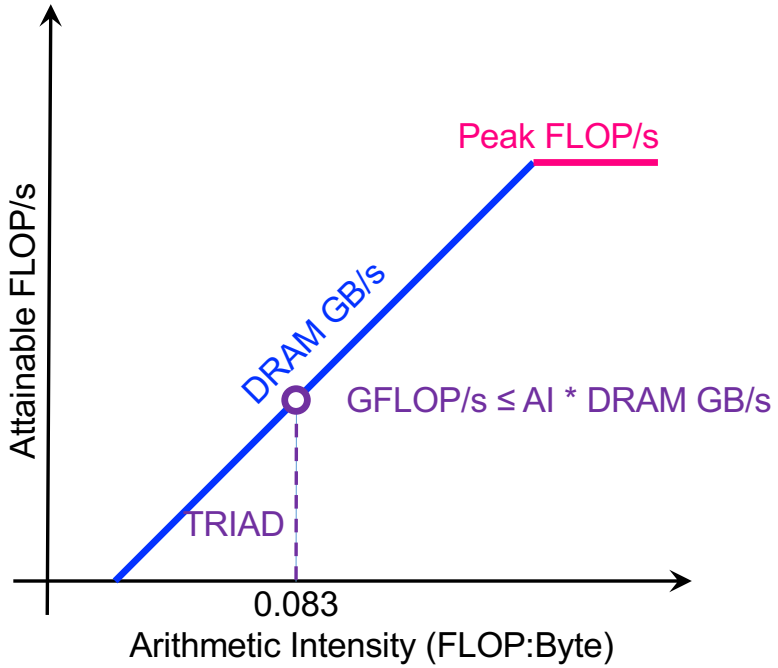
Roofline Example #1 (H)

- Typical machine balance is 5-10 flops per byte…
  - 40-80 flops per double to exploit compute capability
  - Artifact of technology and money
  - **Unlikely to improve**

- Consider STREAM Triad…

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

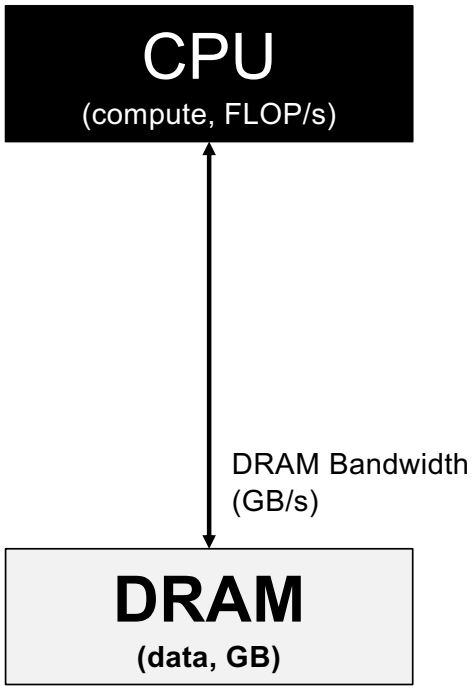
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 flops per byte == Memory bound**



Roofline Example #2

- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - **AI = 0.11 flops per byte (L1)**

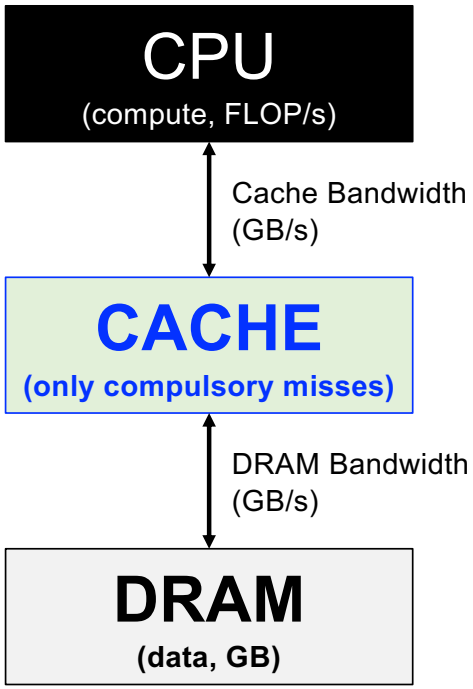
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                     + old[k  ][j  ][i-1]
                     + old[k  ][j  ][i+1]
                     + old[k  ][j-1][i  ]
                     + old[k  ][j+1][i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



Roofline Example #2

- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - Cache can filter all but 1 read and 1 write per point
  - **AI = 0.44 flops per byte**

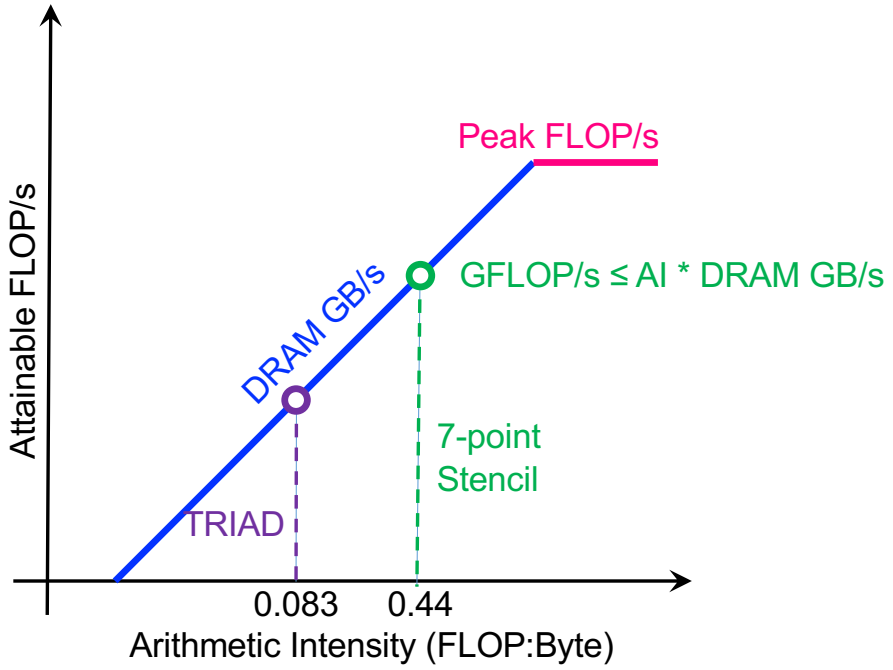
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
 new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                + old[k  ][j  ][i-1]
                + old[k  ][j  ][i+1]
                + old[k  ][j-1][i  ]
                + old[k  ][j+1][i  ]
                +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



Roofline Example #2

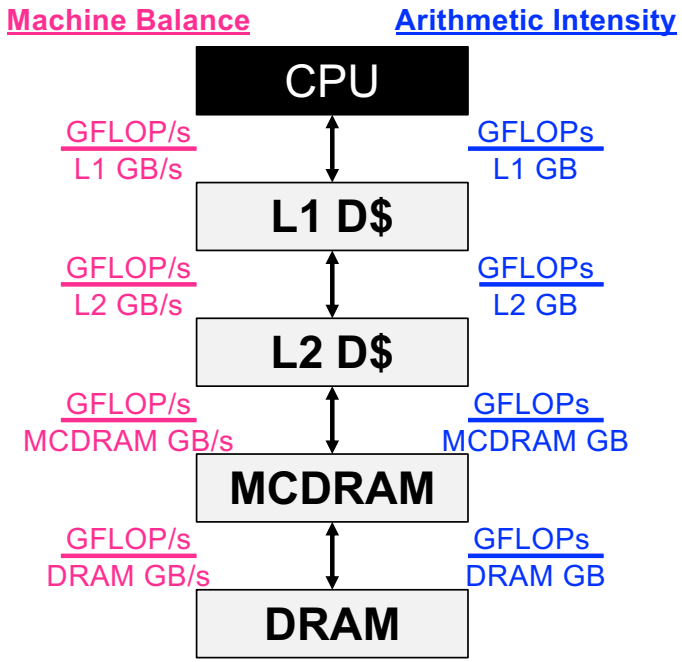
- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - Cache can filter all but 1 read and 1 write per point
  - **AI = 0.44 flops per byte == memory bound,**
  - **but 5x the flop rate**

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



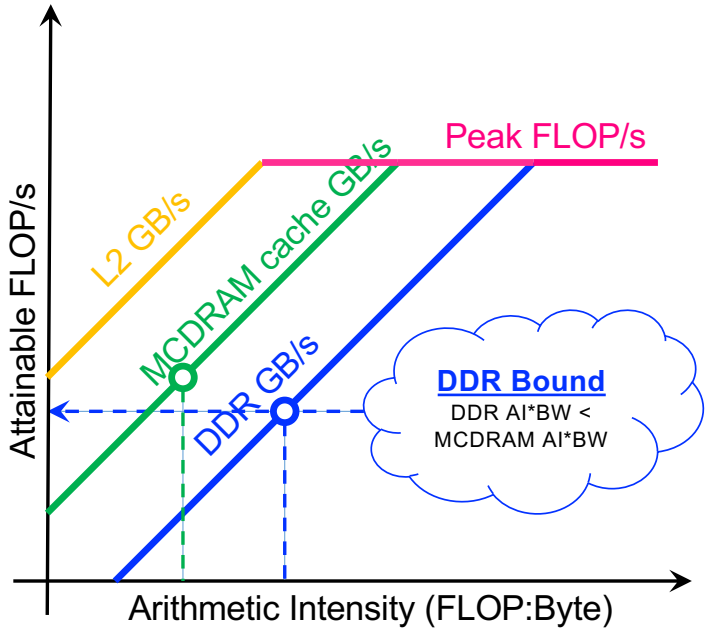
Hierarchical Roofline

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



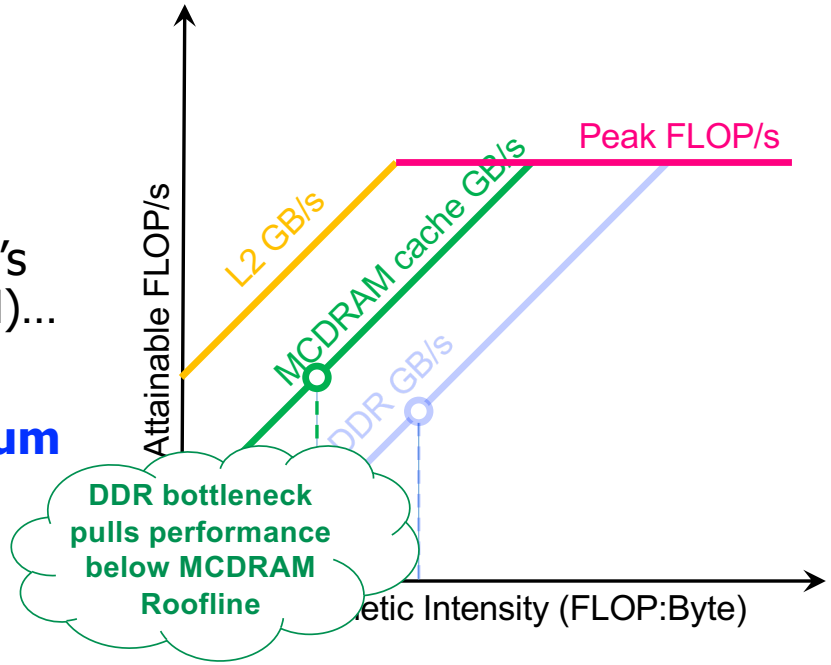
Hierarchical Roofline

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



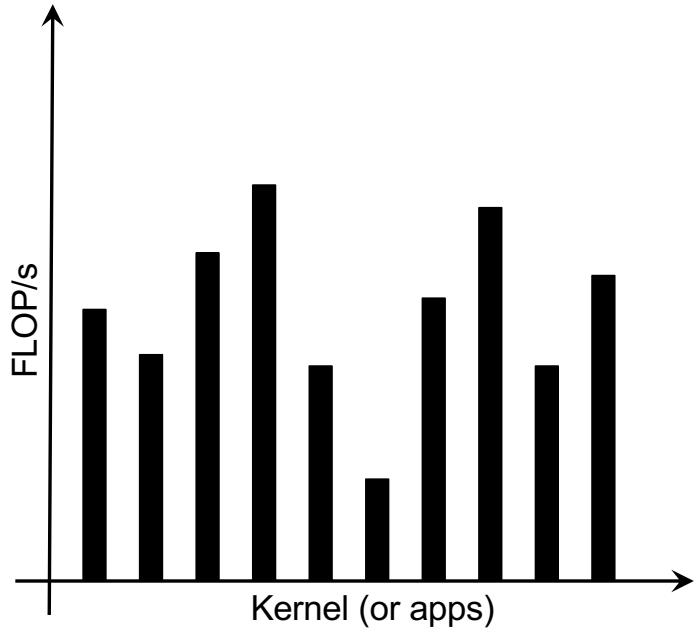
Hierarchical Roofline

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



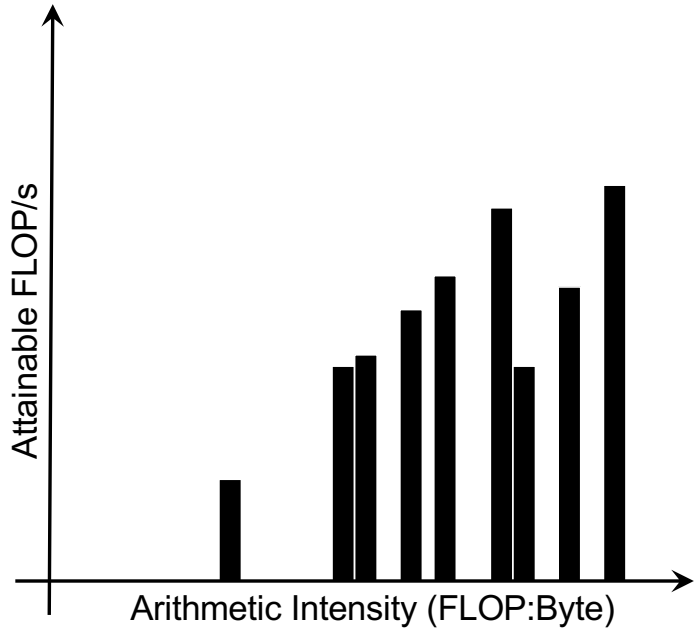
Why is Roofline Useful?

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



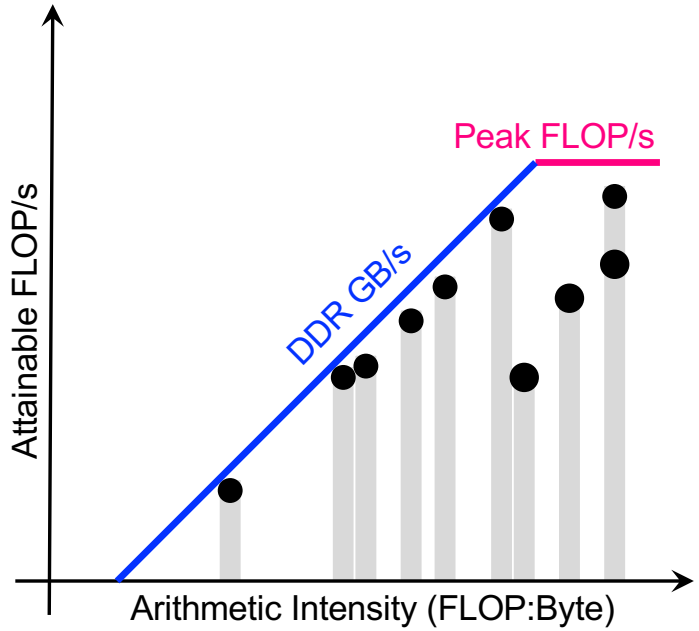
Why is Roofline Useful?

- We can sort kernels by AI …



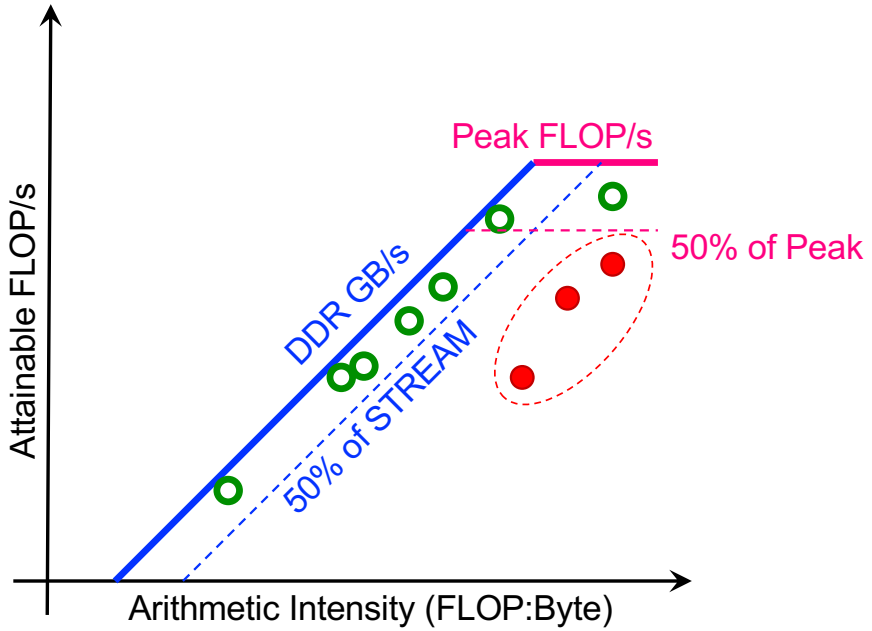
Why is Roofline Useful?

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



Why is Roofline Useful?

- Kernels near the roofline are making good use of computational resources
  - kernels can have low performance (GFLOP/s), but make good use of a machine
  - kernels can have high performance (GFLOP/s), but make poor use of a machine



How Do We Count FLOPs?

Manual Counting

- Go thru each loop nest and count the number of FP operations
- ✓ Works best for deterministic loop bounds
- ✓ or parameterize by the number of iterations (recorded at run time)
- ✗ Not scalable

Perf. Counters

- Read counter before/after
- ✓ More Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✗ Requires privileged access
- ✗ Requires manual instrumentation (+overhead) or full-app characterization
- ✗ Broken counters = garbage
- ✗ May not differentiate FMADD from FADD
- ✗ No insight into special pipelines

Binary Instrumentation

- Automated inspection of assembly at run time
- ✓ Most Accurate
- ✓ FMA-, VL-, and mask-aware
- ✓ Can count instructions by class/type
- ✓ Can detect load imbalance
- ✓ Can include effects from non-FP instructions
- ✓ Automated application to multiple loop nests
- ✗ >10x overhead (short runs / reduced concurrency)

How Do We Measure Data Movement?

Manual Counting

- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
- ✓ Works best for simple loops that stream from DRAM (stencils, FFTs, spare, …)
- ✗ N/A for complex caches
- ✗ Not scalable

Perf. Counters

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

Cache Simulation

- Build a full cache simulator driven by memory addresses
- ✓ Applies to full hierarchy and multicore
- ✓ Can detect load imbalance
- ✓ Automated application to multiple loop nests
- ✗ Ignores prefetchers
- ✗ >10x overhead (short runs / reduced concurrency)

Can we do this differently?

- of course …
- e.g., using **Operational Laws** (what Laws?)
- *Operational Laws* are similar to the elementary laws of motion, for example:

$$d = \frac{1}{2}at^2$$

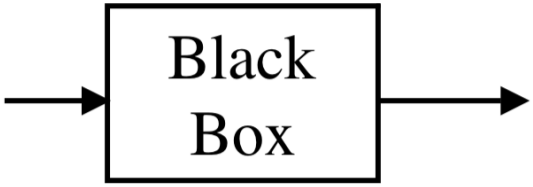
- Notice that distance *d*, acceleration *a*, and time *t* are **operational quantities**. No need to consider them as expected values of random variables or to assume a distribution

Operational Laws

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times
- Identified originally by Buzen (1976) and later extended by Denning and Buzen (1978)
- **Operational ⇒ Directly measured**
- Operationally testable assumptions ⇒ assumptions that can be verified by measurements
  - For example, whether number of job arrivals is equal to the number of completions?
  - This assumption, called job flow balance, is operationally testable
  - A set of observed service times is or is not a sequence of independent random variables is or is not operationally testable

Operational Quantities

- Quantities that can be directly measured during a finite observation period
- T = Observation interval — A_i = #arrivals
- C_i = #completions — B_i = Busy Time



$$\text{Arrival Rate } \lambda_i = \frac{\text{Number of arrivals}}{\text{Time}} = \frac{A_i}{T}$$

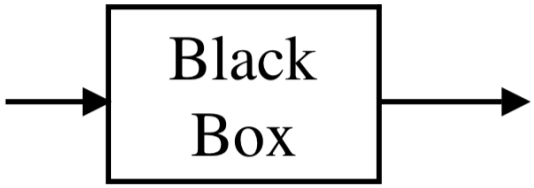
$$\text{Throughput } X_i = \frac{\text{Number of completions}}{\text{Time}} = \frac{C_i}{T}$$

$$\text{Utilization } U_i = \frac{\text{Busy Time}}{\text{Total Time}} = \frac{B_i}{T}$$

$$\text{Mean service time } S_i = \frac{\text{Total time Served}}{\text{Number served}} = \frac{B_i}{C_i}$$

Utilization Law

$$\begin{aligned}\text{Utilization } U_i &= \frac{\text{Busy Time}}{\text{Total Time}} = \frac{B_i}{T} \\ &= \frac{C_i}{T} \times \frac{B_i}{C_i} = \frac{\text{Completions}}{\text{Time}} \times \frac{\text{Busy Time}}{\text{Completions}} \\ &= \text{Throughput} \times \text{Mean Service Time} = X_i S_i\end{aligned}$$



Motivational Example:

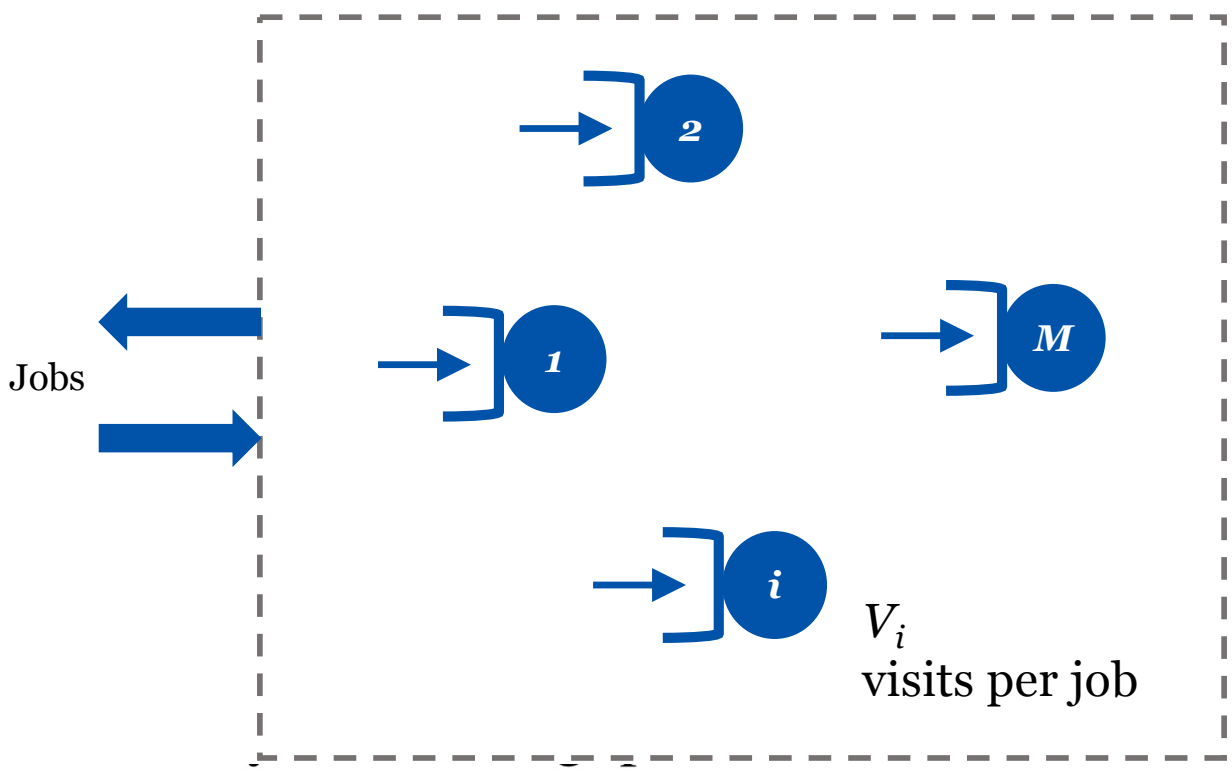
- Consider a network gateway at which the packets arrive at a rate of **125** packets per second and the gateway takes an average of two milliseconds (2ms) to forward them
- Throughput X_i = Exit rate = Arrival rate = 125 packets/second
- Service time Si = 0.002 seconds
- Utilization $U_i = X_i S_i = 125 \times 0.002 = 0.25 =$ ***25%***
- This result is valid **for any** arrival or service process. Even if inter-arrival times and service times to are not IID* random variables with exponential distribution (operational)

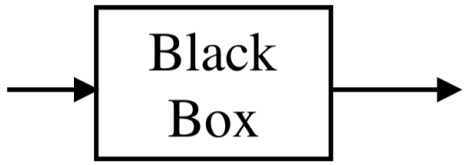
* IID: Independent and Identically Distributed random variables

Forced Flow Law

- Relates the system throughput to individual device throughputs
- In an **open model**, System throughput = #jobs leaving the system per unit time
- In a **closed model**, System throughput = #jobs traversing OUT to IN link per unit time
- If observation period T is such that $A_i = C_i$
  $\Rightarrow$ Device satisfies the assumption of job flow balance
- Each job makes V_i requests for i^{th} device in the system
- $C_i = C_0 V_i$ or $V_i = C_i/C_0$, V_i is called **visit ratio**

Forced Flow Law (cont)





$$\text{System throughput } X = \frac{\text{Jobs completed}}{\text{Total time}} = \frac{C_0}{T}$$

Forced Flow Law (cont)

- Throughput of i^{th} device:

$$\text{Device Throughput } X_i = \frac{C_i}{T} = \frac{C_i}{C_0} \times \frac{C_0}{T}$$

- In other words for throughput:

$$X_i = XV_i$$

- This is the **Forced Flow Law**

Bottleneck Device

- Combining the forced flow law and the utilization law, we get:

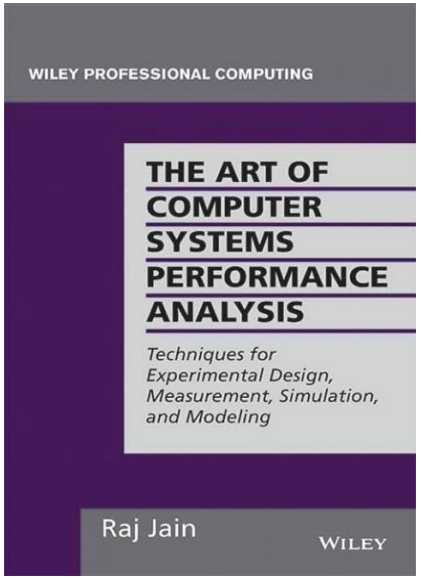
$$\begin{aligned} \text{Utilization of } i^{\text{th}} \text{ device } U_i &= X_i S_i \\ &= X V_i S_i \end{aligned}$$

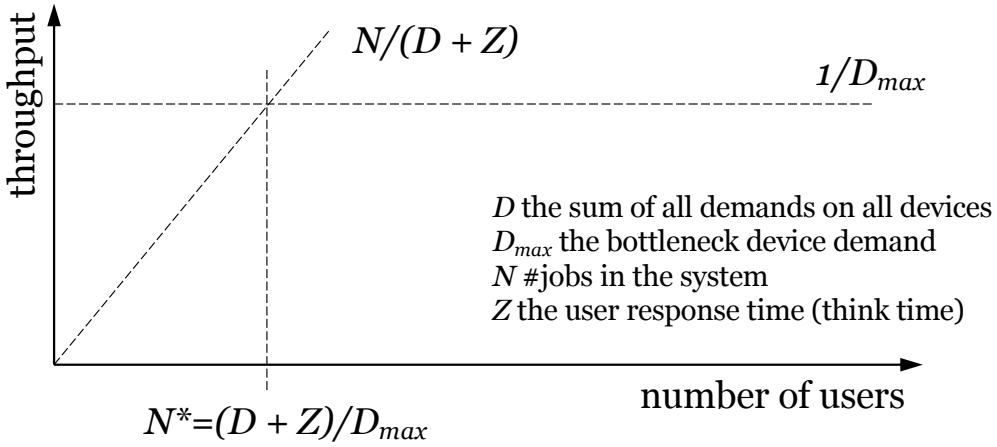
$$U_i = X D_i$$

- Here $D_i = V_i S_i$ is the *total service demand* on the device for all visits of a job
- The device with the highest D_i has the highest utilization and is the **bottleneck device**

And there are more related Laws

Utilization Law:	U_i	$=$	$X_i S_i = X D_i$
Forced Flow Law:	X_i	$=$	$X V_i$
Little's Law:	Q_i	$=$	$X_i R_i$
General Response Time Law:	R	$=$	$\sum_{i=1}^{M} R_i V_i$
Interactive Response Time Law:	R	$=$	$\frac{N}{X} - Z$
Asymptotic Bounds:	R	$\geq$	$max\{D, N D_{max} - Z\}$
	X	$\leq$	$min\{1/D_{max}, N/(D+Z)\}$

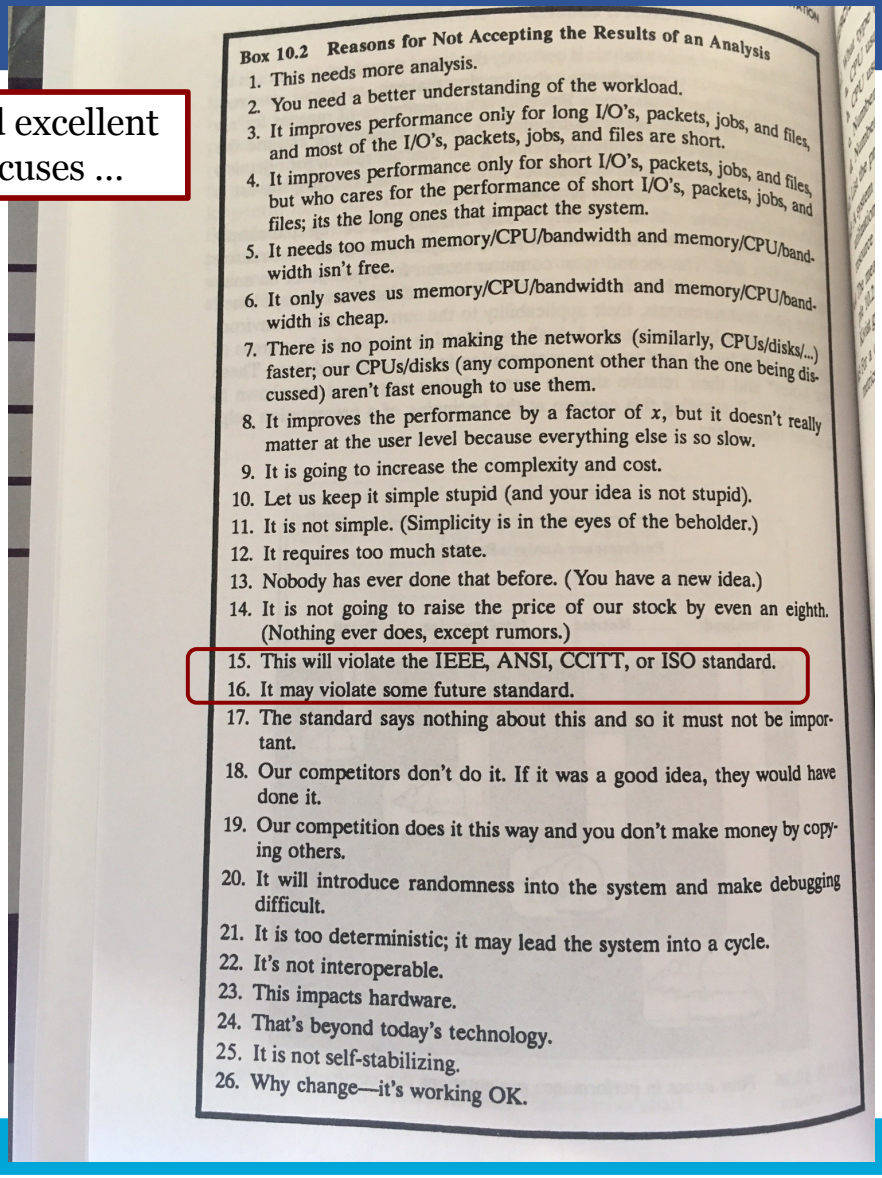




D the sum of all demands on all devices
D_{max} the bottleneck device demand
N #jobs in the system
Z the user response time (think time)

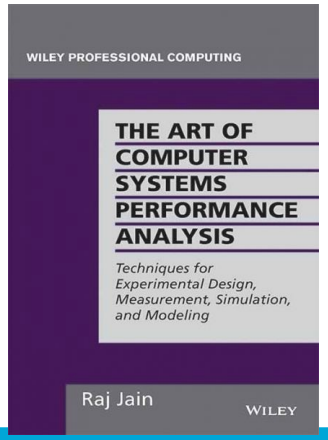
Raj Jain, "The Art of Computer Systems Performance Analysis," Wiley, 1991

Will also find excellent source for excuses …



Box 10.2 Reasons for Not Accepting the Results of an Analysis

1. This needs more analysis.
2. You need a better understanding of the workload.
3. It improves performance only for long I/O's, packets, jobs, and files, and most of the I/O's, packets, jobs, and files are short.
4. It improves performance only for short I/O's, packets, jobs, and files, but who cares for the performance of short I/O's, packets, jobs, and files; its the long ones that impact the system.
5. It needs too much memory/CPU/bandwidth and memory/CPU/bandwidth isn't free.
6. It only saves us memory/CPU/bandwidth and memory/CPU/bandwidth is cheap.
7. There is no point in making the networks (similarly, CPUs/disks/...) faster; our CPUs/disks (any component other than the one being discussed) aren't fast enough to use them.
8. It improves the performance by a factor of x, but it doesn't really matter at the user level because everything else is so slow.
9. It is going to increase the complexity and cost.
10. Let us keep it simple stupid (and your idea is not stupid).
11. It is not simple. (Simplicity is in the eyes of the beholder.)
12. It requires too much state.
13. Nobody has ever done that before. (You have a new idea.)
14. It is not going to raise the price of our stock by even an eighth. (Nothing ever does, except rumors.)
15. This will violate the IEEE, ANSI, CCITT, or ISO standard.
16. It may violate some future standard.
17. The standard says nothing about this and so it must not be important.
18. Our competitors don't do it. If it was a good idea, they would have done it.
19. Our competition does it this way and you don't make money by copying others.
20. It will introduce randomness into the system and make debugging difficult.
21. It is too deterministic; it may lead the system into a cycle.
22. It's not interoperable.
23. This impacts hardware.
24. That's beyond today's technology.
25. It is not self-stabilizing.
26. Why change—it's working OK.

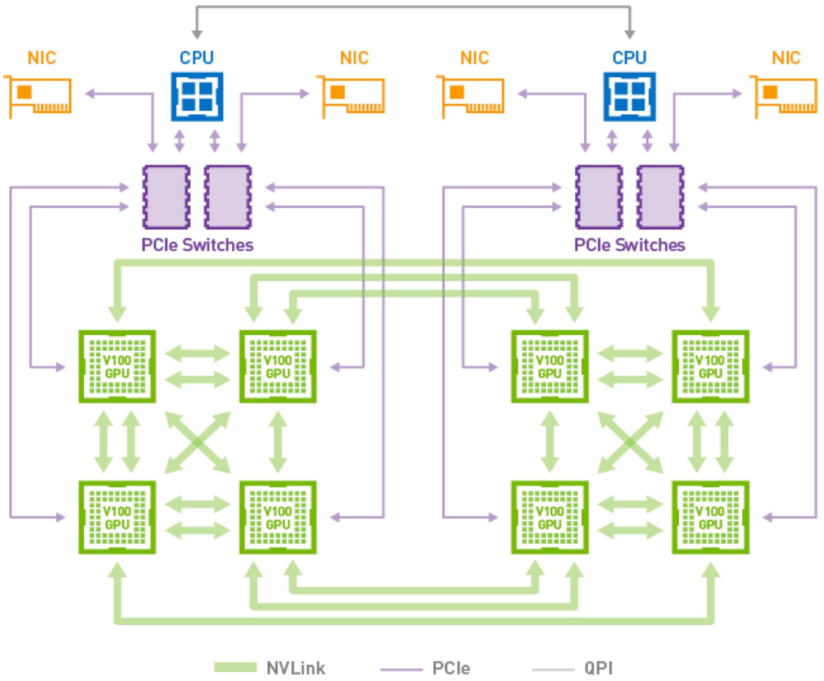


Recommended read!

Raj Jain, "The Art of Computer Systems Performance Analysis," Wiley, 1991

Summary

- Fair performance comparison is tricky (do not cook the numbers)
- Real application level performance matters but is not easy
- Best serial program has not much to do with the best || program

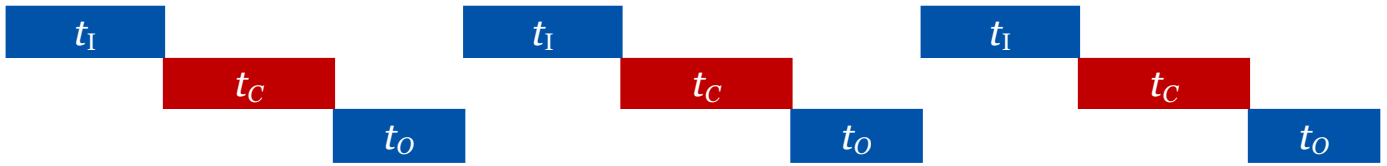


Thank you

university of groningen

Simple Motivational Example

- Assume a repetitive task reading a record (t_I), processing (t_C) it and then storing the results back (t_O)
  - It is irrelevant if t_I and t_O represent I/O or Memory accesses
- Serial execution (no overlap):

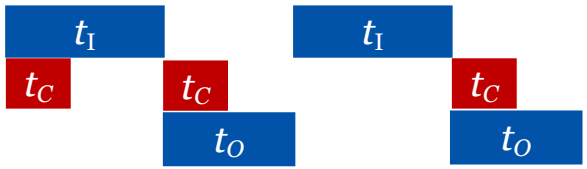


$$Throughput = 1/(t_I + t_C + t_O) \text{ [records/sec]}$$

Simple Motivational Example (cont.)

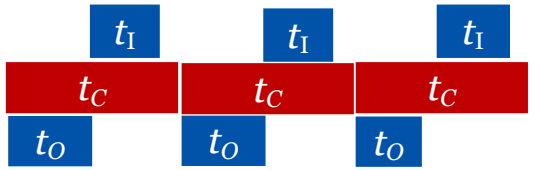
- Assume a repetitive task reading a record (t_I), processing (t_C) it and then storing the results back (t_O)
  - Irrelevant if t_I and t_O represent I/O or Memory access
- Two-way restricted parallel (C-I or O-C) (degree of overlap 2):

$t_C < t_I + t_O$



$Throughput = 1/(t_I + t_O)$

$t_C \geq t_I + t_O$



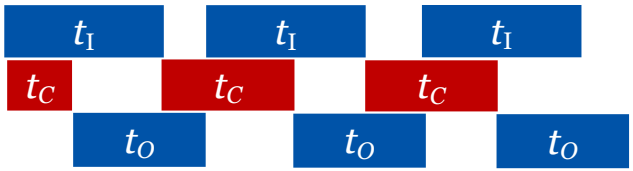
$Throughput = 1/(t_C)$

university of groningen

Simple Motivational Example (cont.)

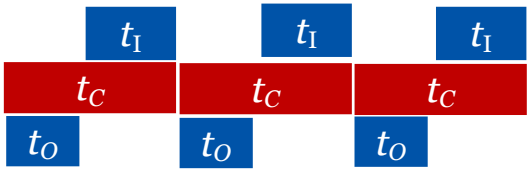
- Assume a repetitive task reading a record (t_I), processing (t_C) it and then storing the results back (t_O)
  - Irrelevant if t_I and t_O represent I/O or Memory access
- Two-way parallel (C-I or O-C or I-O) (degree of overlap 2):

$t_C < t_I + t_O$



$Throughput = 2/(t_I + t_C + t_O)$

$t_C \geq t_I + t_O$



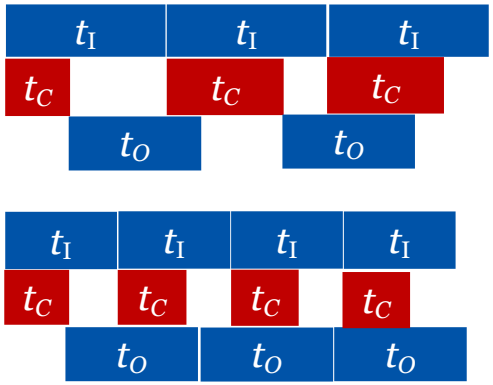
$Throughput = 1/(t_C)$

university of groningen

Simple Motivational Example (cont.)

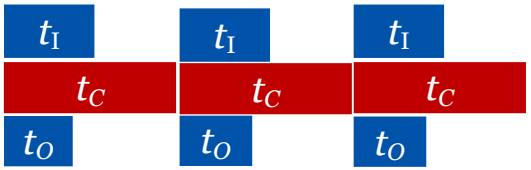
- Assume a repetitive task reading a record (t_I), processing (t_C) it and then storing the results back (t_O)
- Three-way parallel (I-C-O) (degree of overlap 3):

$t_C < t_I + t_O$



$Throughput = 1/(\max(t_I, t_O))$

$t_C \geq t_I + t_O$



$Throughput = 1/(t_C)$

university of groningen

Simple Motivational Example (cont.)

- The presented model is rough and quite optimistic
- Average times are used
  - Suits well streaming with FIFOs used to balance data moves
- Provides an upper bound for what is achievable for your application on a given computing platform
  - t_I, t_C and t_O summarize platform and application properties
  - They are determined by platform capabilities and application needs
    - How much data is moved and how much computation for each data element
- The model's simplicity enables early strategic decisions and will prevent you of committing on impossible targets

university of groningen

Iron Law Example

- Machine A: clock 1ns, CPI 2.0, for program x
- Machine B: clock 2ns, CPI 1.2, for program x

- Which is faster and how much?

  Time/Program = instr/program x cycles/instr x sec/cycle

  Time(A) = N x 2.0 x 1 = 2N

  Time(B) = N x 1.2 x 2 = 2.4N

  Compare: Time(B)/Time(A) = 2.4N/2N = 1.2

- So, Machine A is 20% faster than Machine B for this program

university of groningen

Iron Law Example

Keep clock(A) @ 1ns and clock(B) @2ns

For equal performance, if CPI(B)=1.2, what is CPI(A)?

Time(B)/Time(A) = 1 = (Nx2x1.2)/(Nx1xCPI(A))

CPI(A) = 2.4

university of groningen

Iron Law Example

Keep CPI(A)=2.0 and CPI(B)=1.2

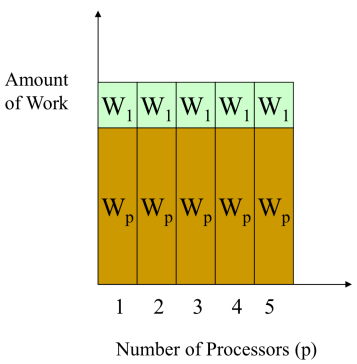
For equal performance, if clock(B)=2ns, what is clock(A)?

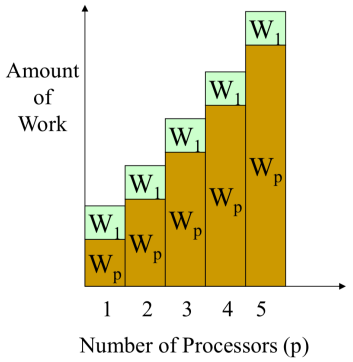
Time(B)/Time(A) = 1 = (N x 2.0 x clock(A))/(N x 1.2 x 2)

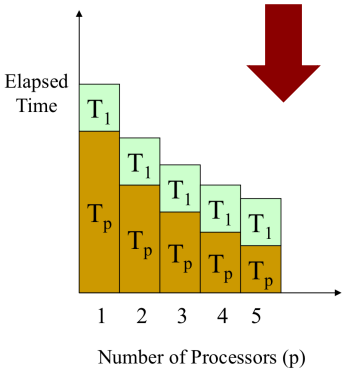
clock(A) = 1.2ns

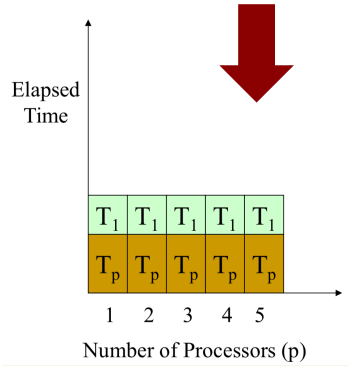
university of groningen

Both Laws Together









Amdhal's view (fixed problem size)

Gustaffson's view (scalable problem size)

Amdhal (v.2):

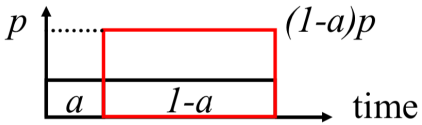
$$\alpha = \frac{t_s}{t_s + t_p}$$

$T_p = \alpha \cdot T_{base} + (1-\alpha) \cdot T_{base} / p$

where, α is the fraction of the serial code and p – the speedup factor of the parallel portion

$T_p = (\alpha + (1-\alpha) / p) \cdot T_s$

$S_p = T_s / T_p = 1/(\alpha + (1-\alpha)/p)$; $\lim_{\infty} = 1/\alpha$



Gustaffson:

S_p = Work (p) / Work (1) =

$= (\alpha \cdot W + (1-\alpha) \cdot p \cdot W)/W = \alpha + (1-\alpha) \cdot p$

linear speedup is assumed

With α = **0.1** (10% serial code)
Amdahl's speedup is maximal **10**,
while Gustaffson claims **0.1 + 0.9·p**

university of groningen

The Roofline Model (Arithmetic Intensity)

```
for (i=0; i < N;
i=i+1)
   a[i] = b[i]+c[i]
```

one (1) ADD
two (2) LOADs (8 Bytes)
one (1) WRITE (8 Bytes)
AI = 1/(2*8+8) = 1/24*

```
for (i=0; i < N; i=i+1)
   a[i] = b[i]*c[i]+b[i]
```

one (1) ADD
one (1) MUL
two (2) LOADs (8 Bytes)
one (1) WRITE (8 Bytes)
AI = 2/(2*8+8) = 1/12*

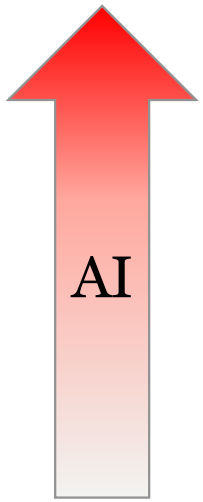
```
for (i=0; i < N; i=i+1) {
   I1 = A_offset[i]; I2 = A_offset[i+1];
   sum = 0.0;
   for (j = 0; j < (I2-I1); ++j)
      sum += A[I1+j] * x[col_index[I2+j]];
   y[i] = sum;
}
```

one (1) ADD
one (1) MUL
two (2) LOADs (8 Bytes)
one (1) LOAD (4 Bytes)
one (1) WRITE (8 Bytes)
AI = 2/(2*8+4+8) = 1/14

* on cache-based systems kernel would actually require an extra read for `a[]` (because of write-allocate traffic) leading to even lower AI

university of groningen

The Roofline Model (Arithmetic Intensity, 2)

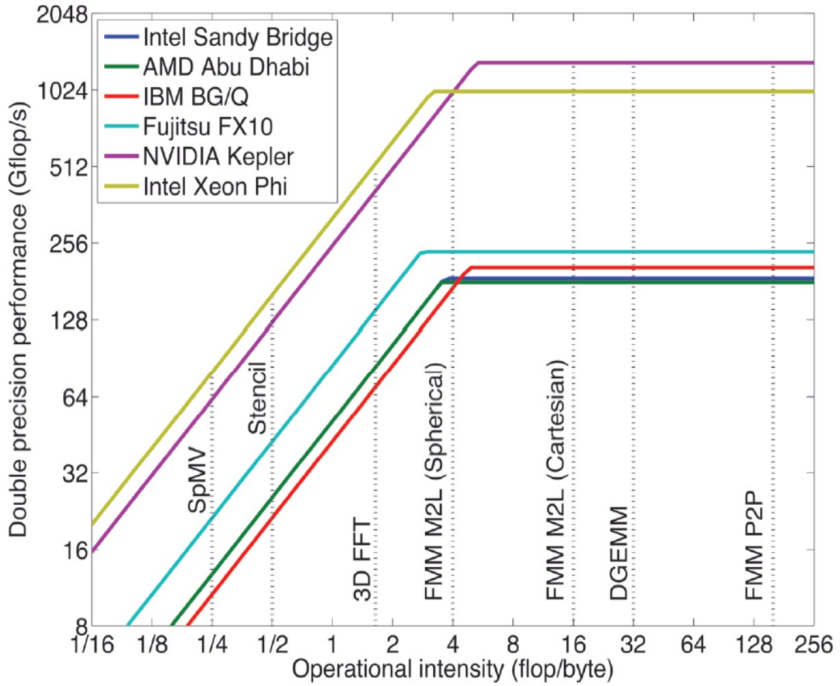


Particle Methods

BLAS L3

FFT Stencil

BLAS L1, SpMv



- New architectures with decreased machine balance
  - the point where the bandwidth roof meets the ceiling moves to the right
- More and more algorithms are going to find themselves memory bound
- Even DGEMM can run into trouble depending on the blocking factor chosen