CS 294-73
Software Engineering for Scientific Computing

Lecture 9: Performance on cache-based systems

Slides from James Demmel and Kathy Yelick

Motivation

- Most applications run at < 10% of the "peak" performance of a system
  - Peak is the maximum the hardware can physically execute
- Much of this performance is lost on a single processor, i.e., the code running on one processor often runs at only 10-20% of the processor peak
- Most of the single processor performance loss is in the memory system
  - Moving data takes much longer than arithmetic and logic

- To understand this, we need to look under the hood of modern processors
  - For today, we will look at only a single "core" processor
  - These issues will exist on processors within any parallel computer

Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case study: Matrix Multiplication
- Roofline model.

Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
- Operations include
  - Read and write into very fast memory called registers
  - Arithmetic and other logical operations on registers
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into "obvious" lower level instructions

A = B + C ⇒

Read address(B) to R1
Read address(C) to R2
R3 = R1 + R2
Write R3 to Address(A)

  - Hardware executes instructions in order specified by compiler
- *Idealized Cost*
  - Each operation has roughly the same cost

    (read, write, add, multiply, etc.)

Uniprocessors in the Real World

- Real processors have
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs
  - parallelism
    - multiple "functional units" that can run in parallel
    - different orders, instruction mixes have different costs
  - pipelining
    - a form of parallelism, like an assembly line in a factory
- Why is this your problem?
    - In theory, compilers and hardware "understand" all this and can optimize your program; in practice they don't.
    - They won't know about a different algorithm that might be a much better "match" to the processor

In theory there is no difference between theory and practice.
But in practice there is. -J. van de Snepscheut

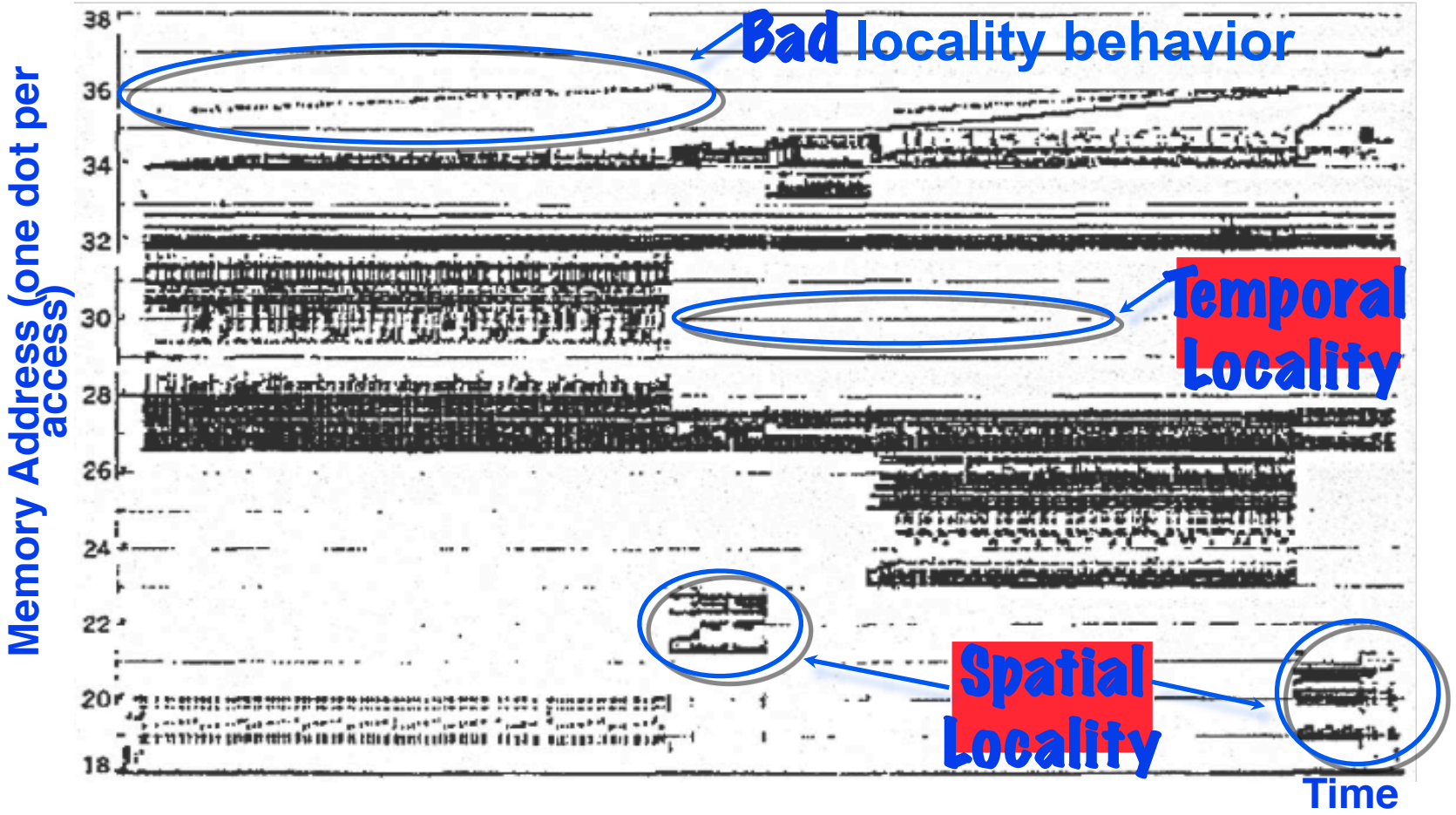
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Temporal and spatial locality
  - Basics of caches
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case study: Matrix Multiplication
- Roofline Model

Approaches to Handling Memory Latency

- Bandwidth has improved more than latency
  - 23% per year vs 7% per year
- Approach to address the memory latency problem
  - Eliminate memory operations by saving values in small, fast memory (cache) and reusing them
    - **need temporal locality in program**
  - Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk
    - **need spatial locality in program**
  - Take advantage of better bandwidth by allowing processor to issue multiple reads to the memory system at once
    - **concurrency in the instruction stream, e.g. load whole array, as in vector processors; or prefetching**
  - Overlap computation & memory operations
    - **prefetching**

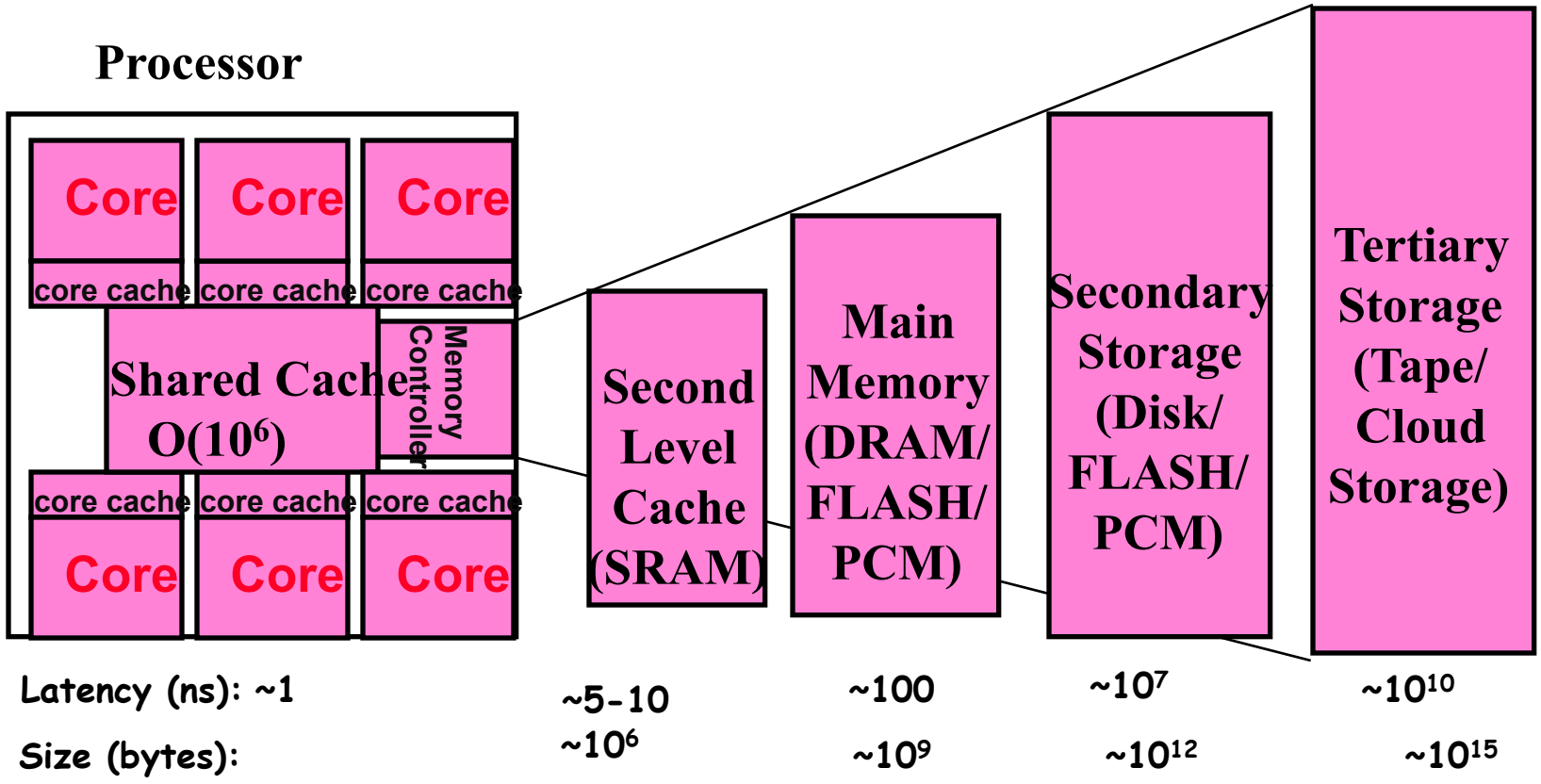
Programs with locality cache well ...



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Cache Basics

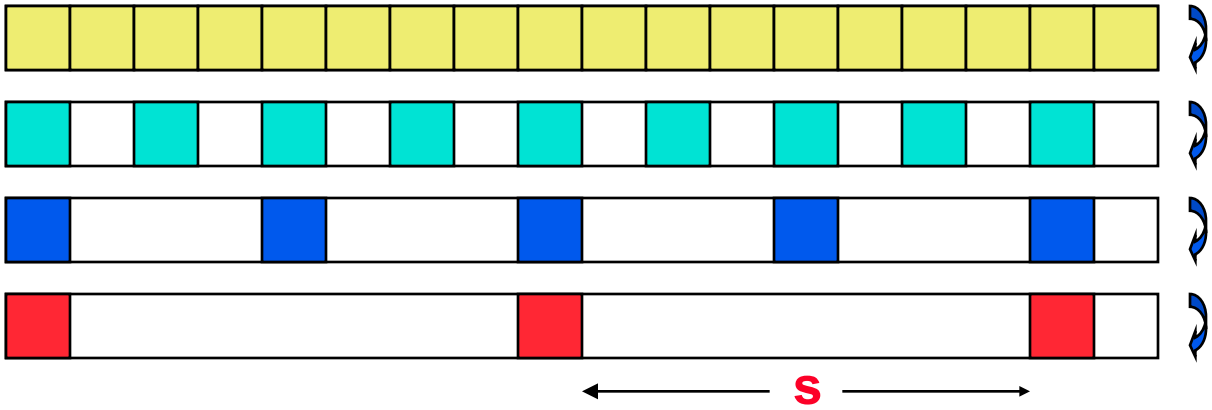
- Cache is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- Cache hit: in-cache memory access—cheap
- Cache miss: non-cached memory access—expensive
  - Need to access next, slower level of cache
- Cache line length: # of bytes loaded together in one entry
  - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are
- Associativity
  - direct-mapped: only 1 address (line) in a given range in cache
    - Data stored at address xxxxx1101 stored at cache location 1101, in 16 word cache
  - *n*-way: $n \geq 2$ lines with different addresses can be stored
    - Example (2-way): addresses xxxxx1100 can be stored at cache location 1101 or 1100.

Why Have Multiple Levels of Cache?

- On-chip vs. off-chip
  - **On-chip caches are faster, but limited in size**
- A large cache has delays
  - **Hardware to check longer addresses in cache takes more time**
  - **Associativity, which gives a more general set of data in cache, also takes more time**
- Some examples:
  - **Cray T3E eliminated one cache to speed up misses**
  - **IBM uses a level of cache as a "victim cache" which is cheaper**
- There are other levels of the memory hierarchy
  - **Register, pages (TLB, virtual memory), … (Page (memory))**
  - **And it isn't always a hierarchy**

Experimental Study of Memory (Membench)

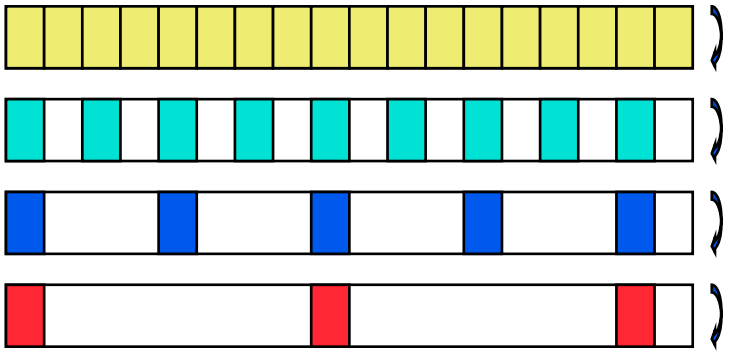
- Microbenchmark for memory system performance

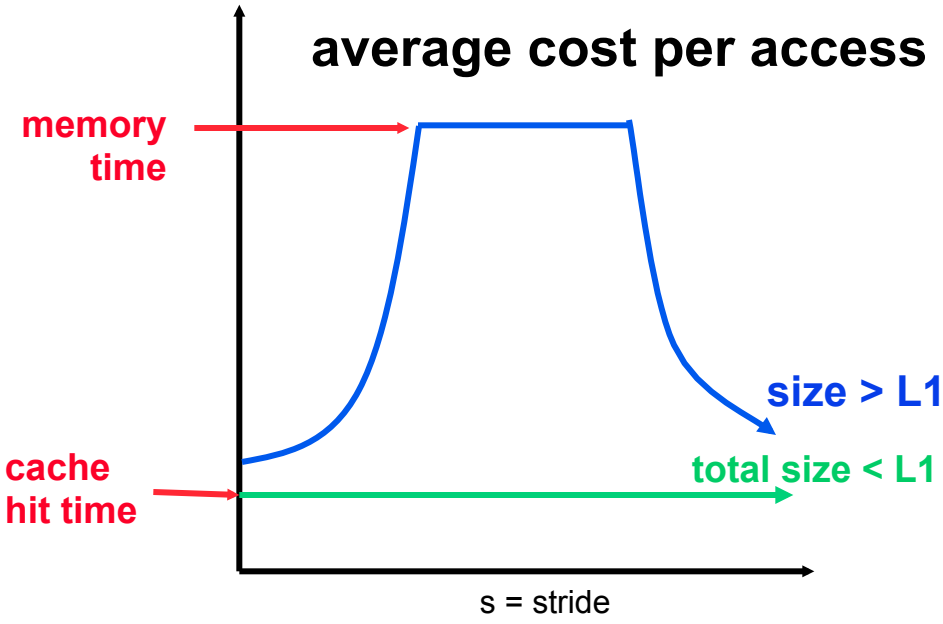


- **for array A of length L from 4KB to 8MB by 2x**
  for stride s from 4 Bytes (1 word) to L/2 by 2x
  time the following loop
  (repeat many times and average)
  for i from 0 to L by s
  load A[i] from memory (4 Bytes)

1 experiment

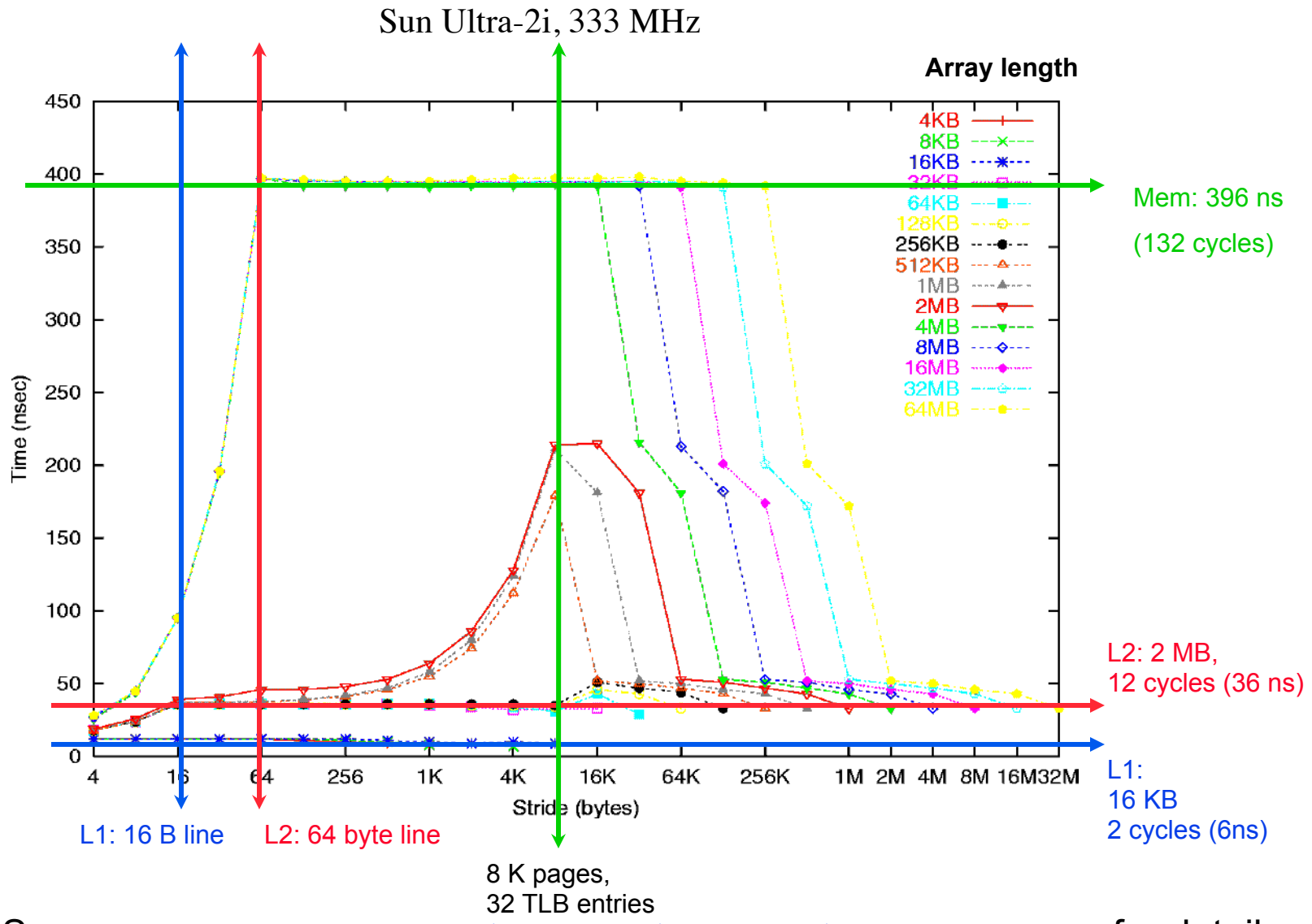
Membench: What to Expect





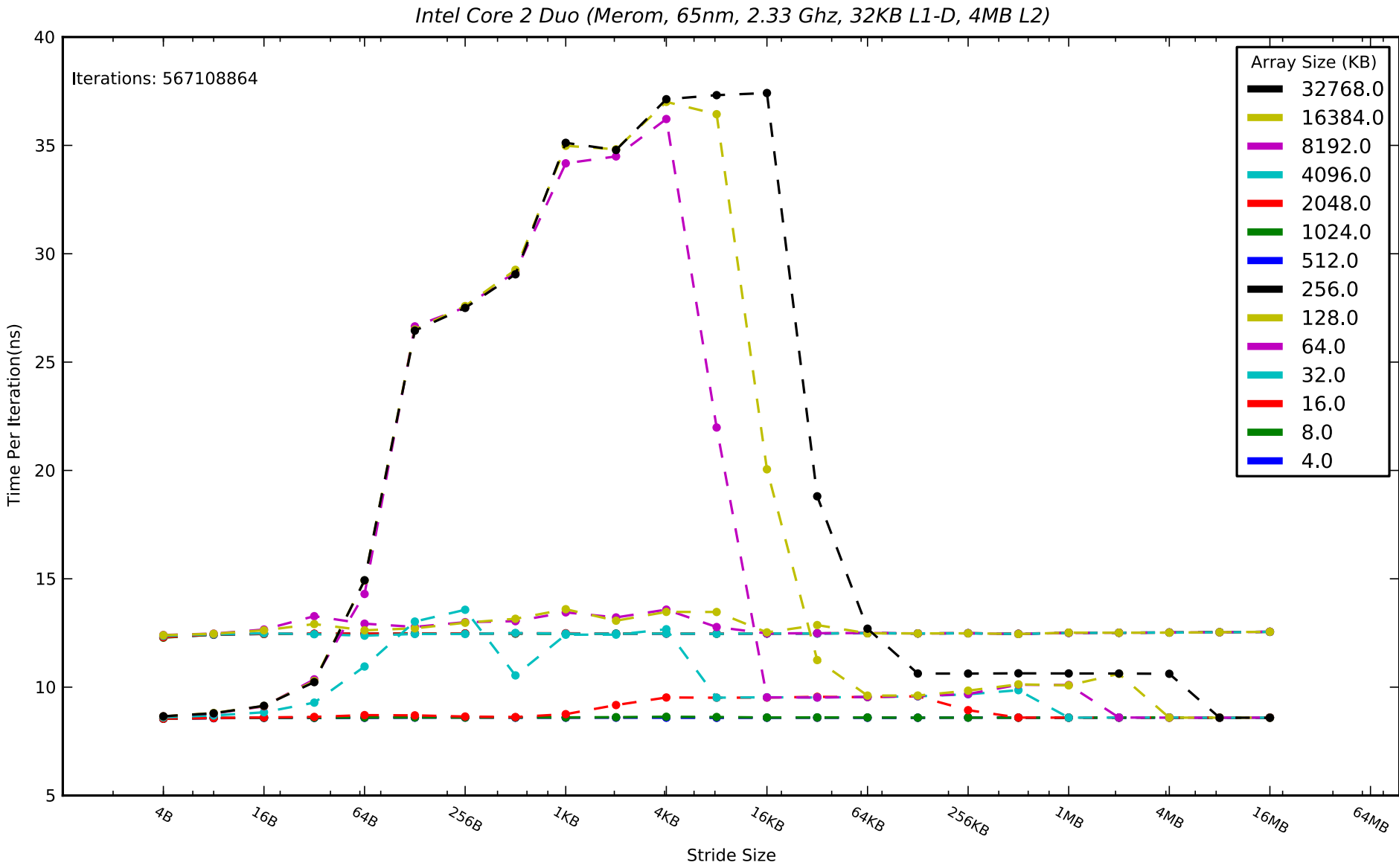
- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most ¼ miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
  - Picture assumes only one level of cache
  - Values have gotten more difficult to measure on modern procs

Memory Hierarchy on a Sun Ultra-2i



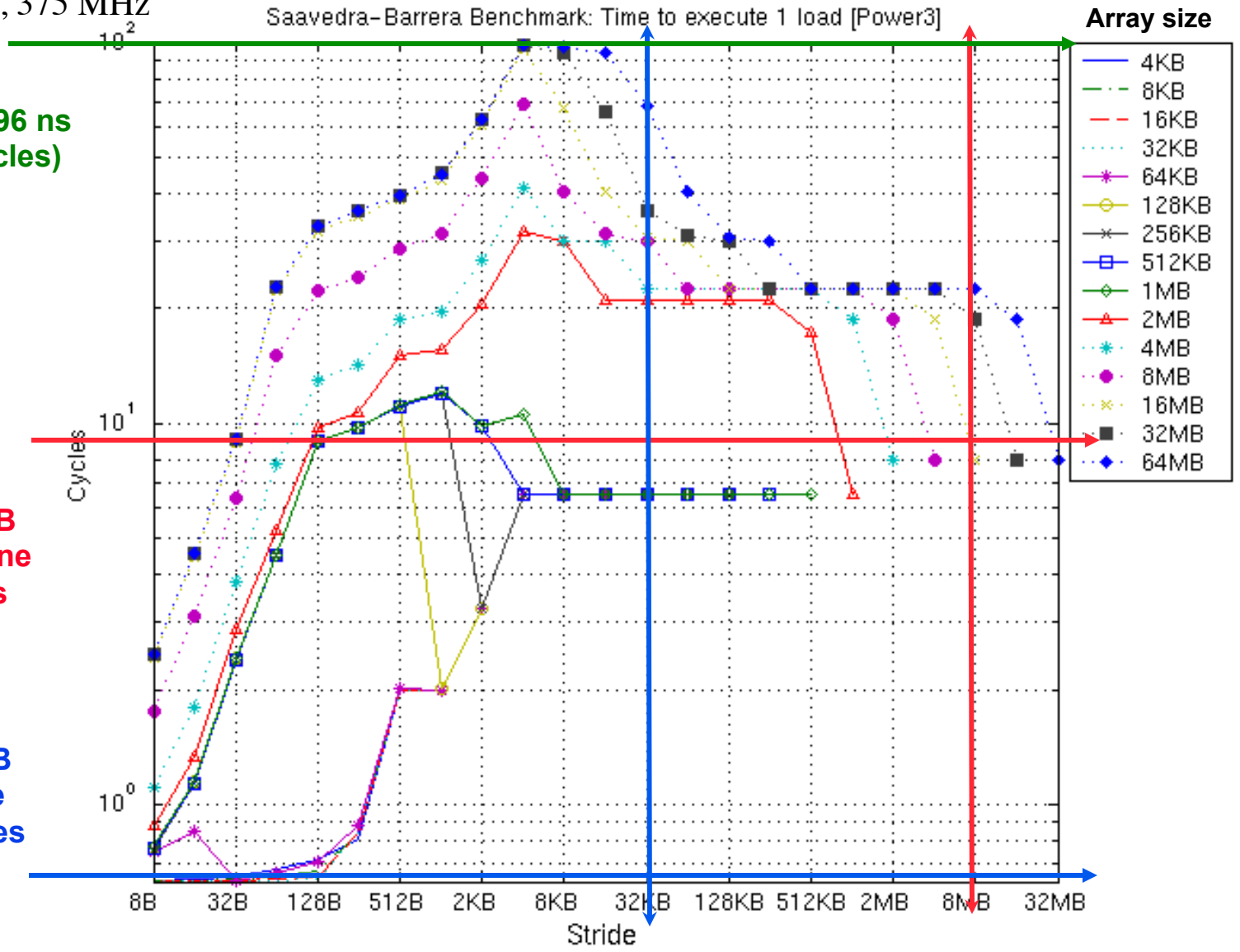
See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details

Memory Hierarchy on an Intel Core 2 Duo



Memory Hierarchy on a Power3 (Seaborg)

Power3, 375 MHz



Mem: 396 ns (132 cycles)

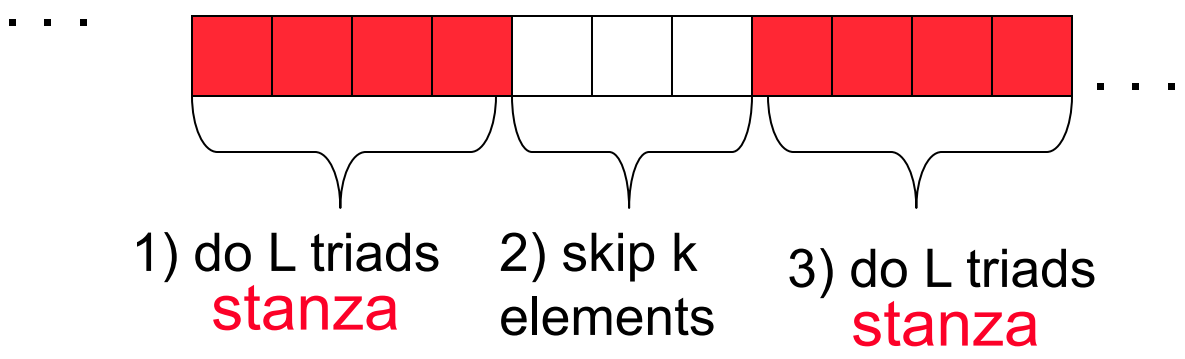
L2: 8 MB
128 B line
9 cycles

L1: 32 KB
128B line
.5-2 cycles

Stanza Triad

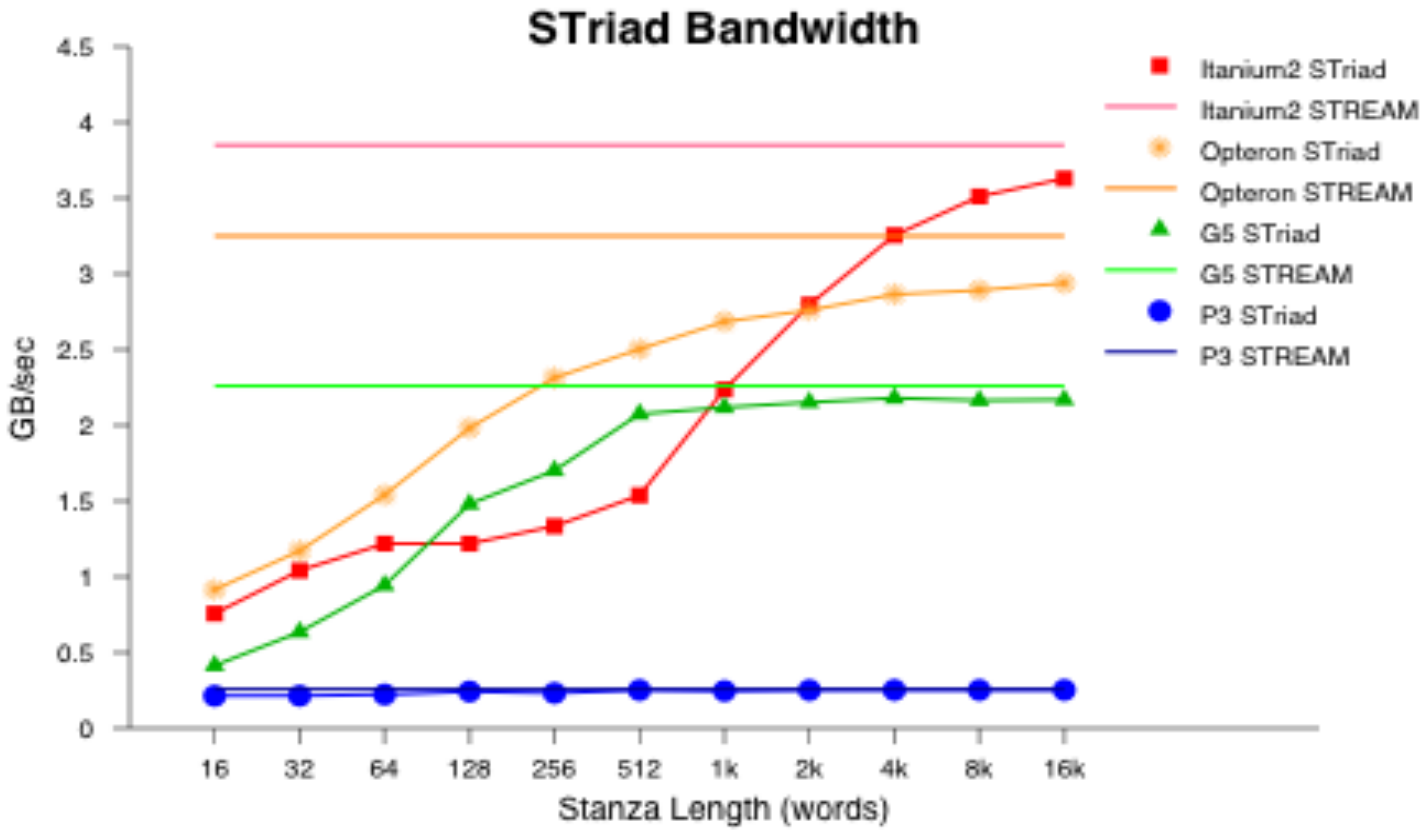
- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- Stanza (L) is the length of a unit stride run

```
while i < arraylength
  for each L element stanza
    A[i] = scalar * X[i] + Y[i]
  skip k elements
```



Source: Kamil et al, MSP05

Stanza Triad Results



- This graph (x-axis) starts at a cache line size (>=16 Bytes)
- If cache locality was the only thing that mattered, we would expect
  - Flat lines equal to measured memory peak bandwidth (STREAM) as on Pentium3
- Prefetching gets the next cache line (pipelining) while using the current one
  - This does not “kick in” immediately, so performance depends on L

Lessons

- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance significantly
  - To write fast programs, need to consider architecture
  - We would like simple models to help us design efficient algorithms

- We will illustrate with a common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

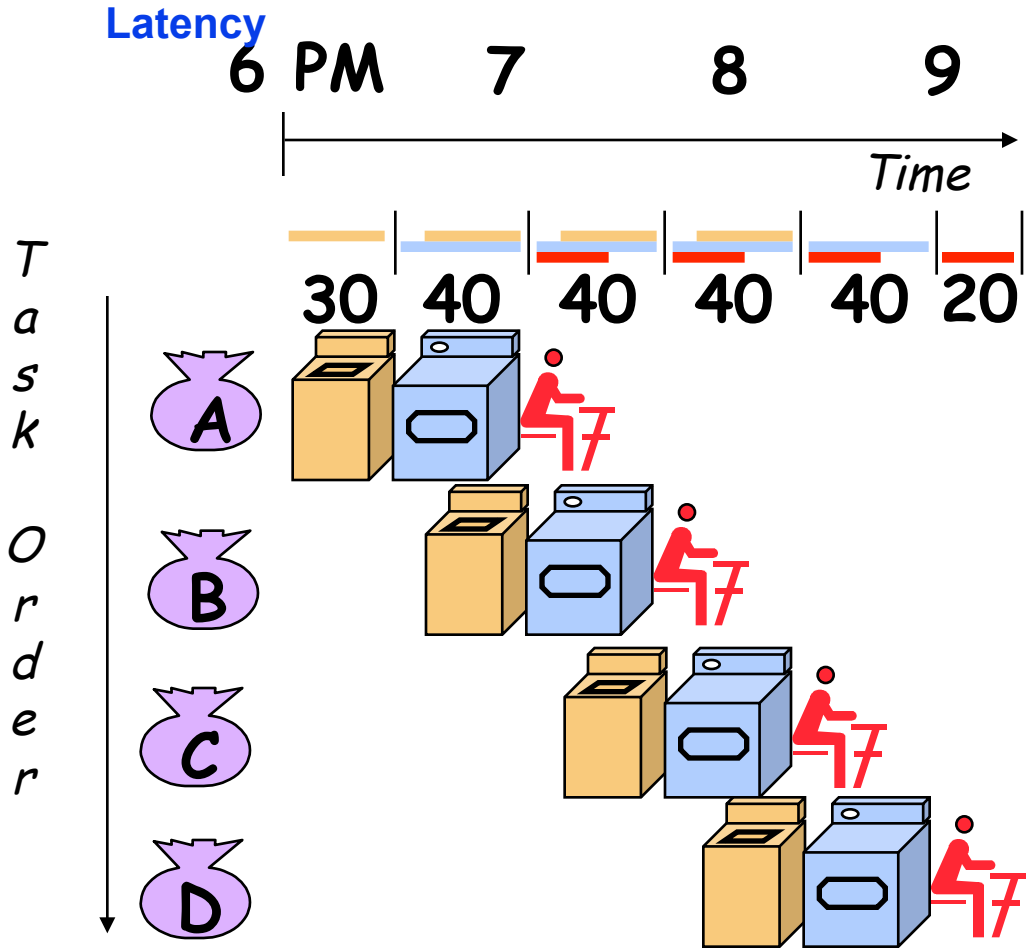
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
  - Hidden from software (sort of)
  - Pipelining
  - SIMD units
- Case study: Matrix Multiplication
- Roofline Model

What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry

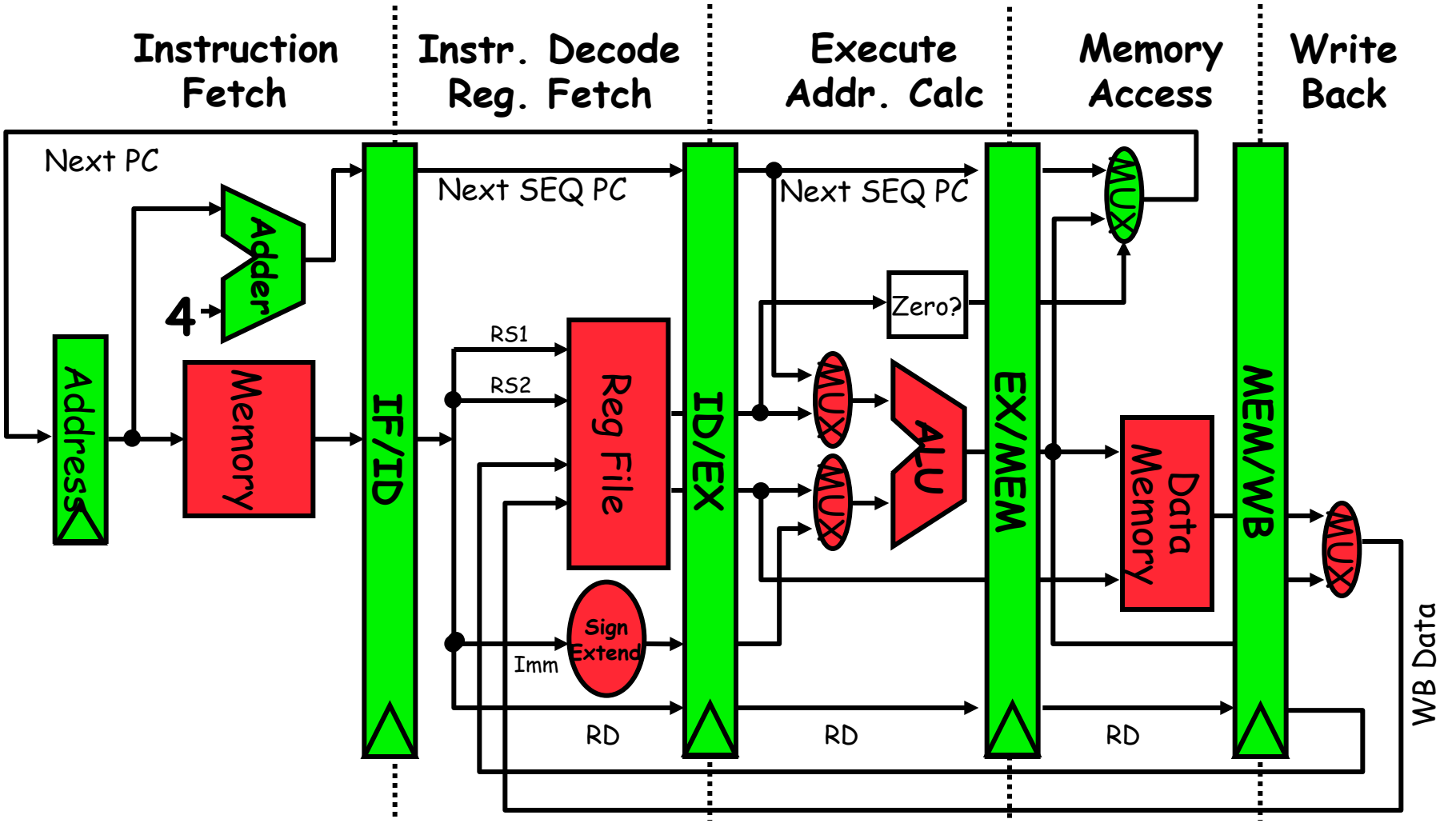
wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
- BW = 4/6 l/h w/o pipelining
- BW = 4/3.5 l/h w pipelining
- BW <= 1.5 l/h w pipelining, more total loads
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipe stages

Example: 5 Steps of MIPS Datapath

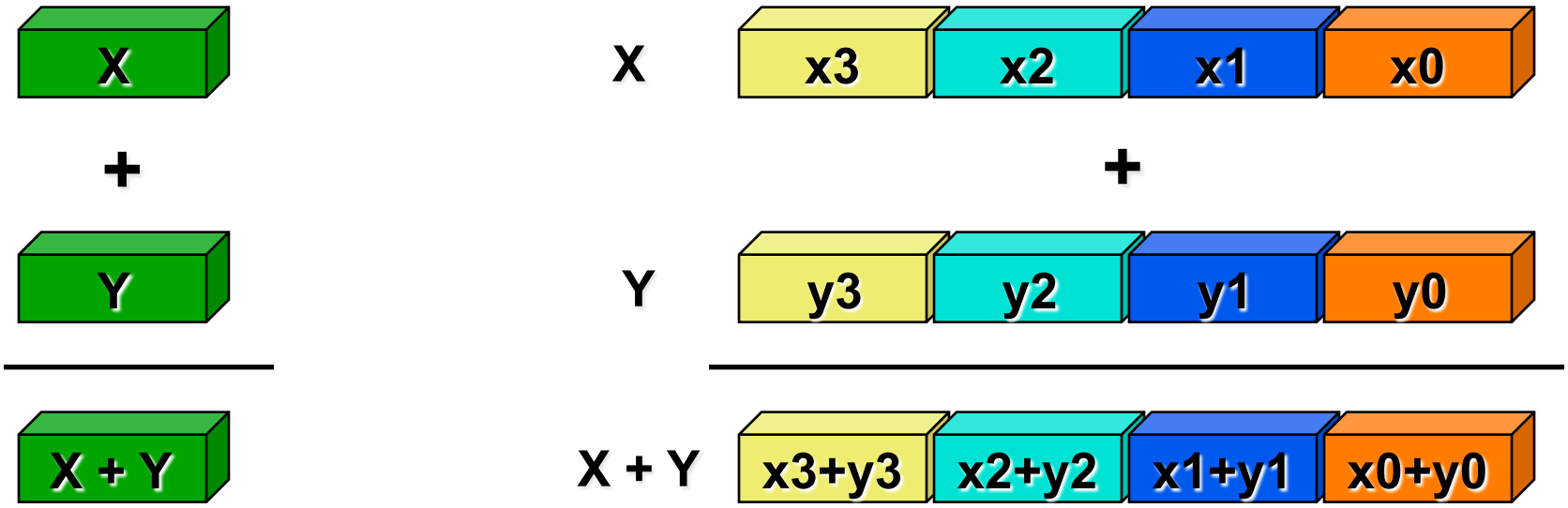
Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy



- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle

SIMD: Single Instruction, Multiple Data

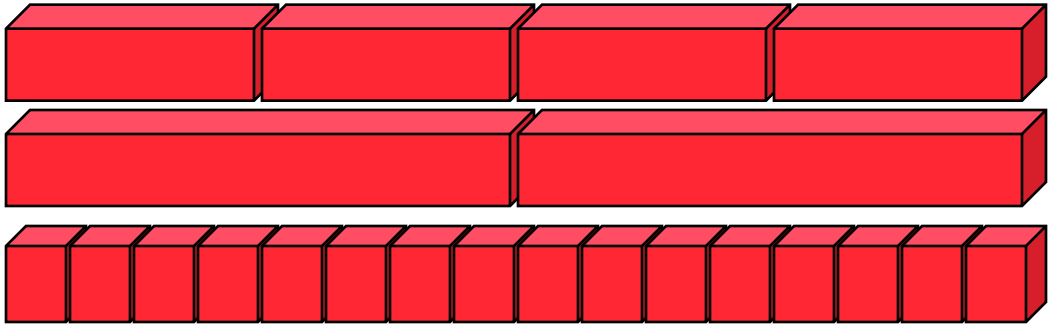
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



4x floats

2x doubles

16x bytes

- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

What does this mean to you?

- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:

    x = y + c * z

    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Outline

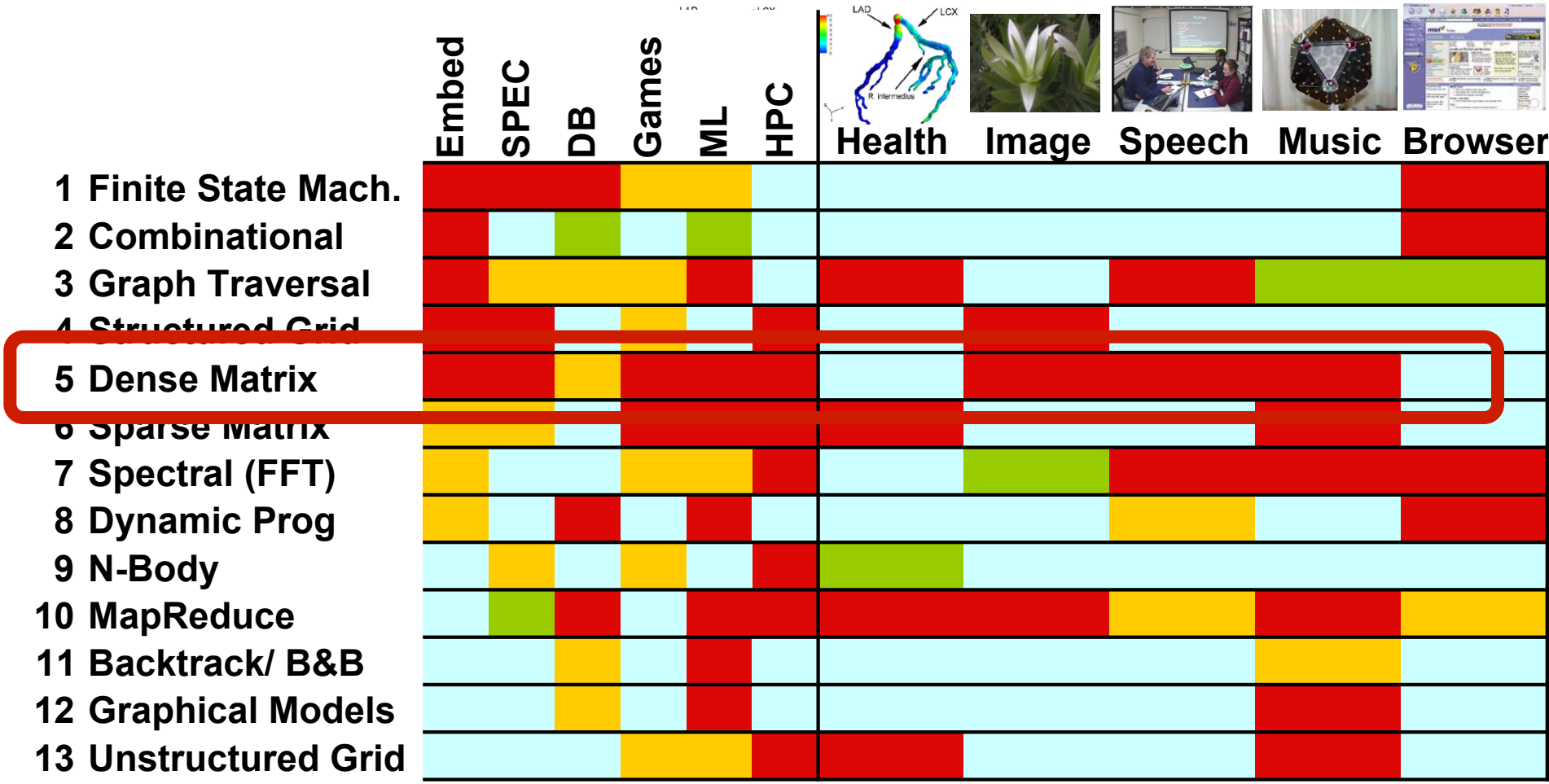
- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Simple cache model
  - Warm-up: Matrix-vector multiplication
  - *(continued next time)*
- Roofline Model

Why Matrix Multiplication?

- An important kernel in many problems
  - Appears in many linear algebra algorithms
    - Bottleneck for dense linear algebra
  - One of the 7 motifs
  - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods
(Red Hot → Blue Cool)



Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A(i,j) at A+i*n+j
  - recursive (later)

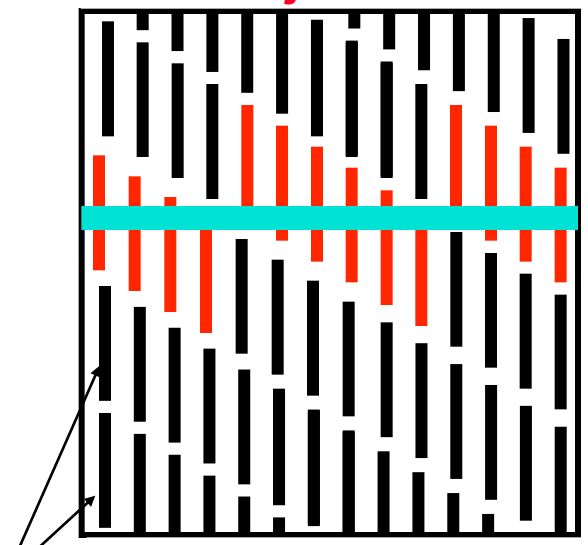
Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Column major matrix in memory



cachelines

Blue row of matrix is stored in red cachelines

- Column major (for now)

Figure source: Larry Carter, UCSD

Modeling Matrix-Vector Multiplication

- Compute time for nxn = 1000x1000 matrix
- Time
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$ (q = #flops/#memory accesses)
  - $= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$
- For t_f and t_m, using data from R. Vuduc's PhD (pp 351-3)
  - http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf
  - For t_m use minimum-memory-latency / words-per-cache-line

	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

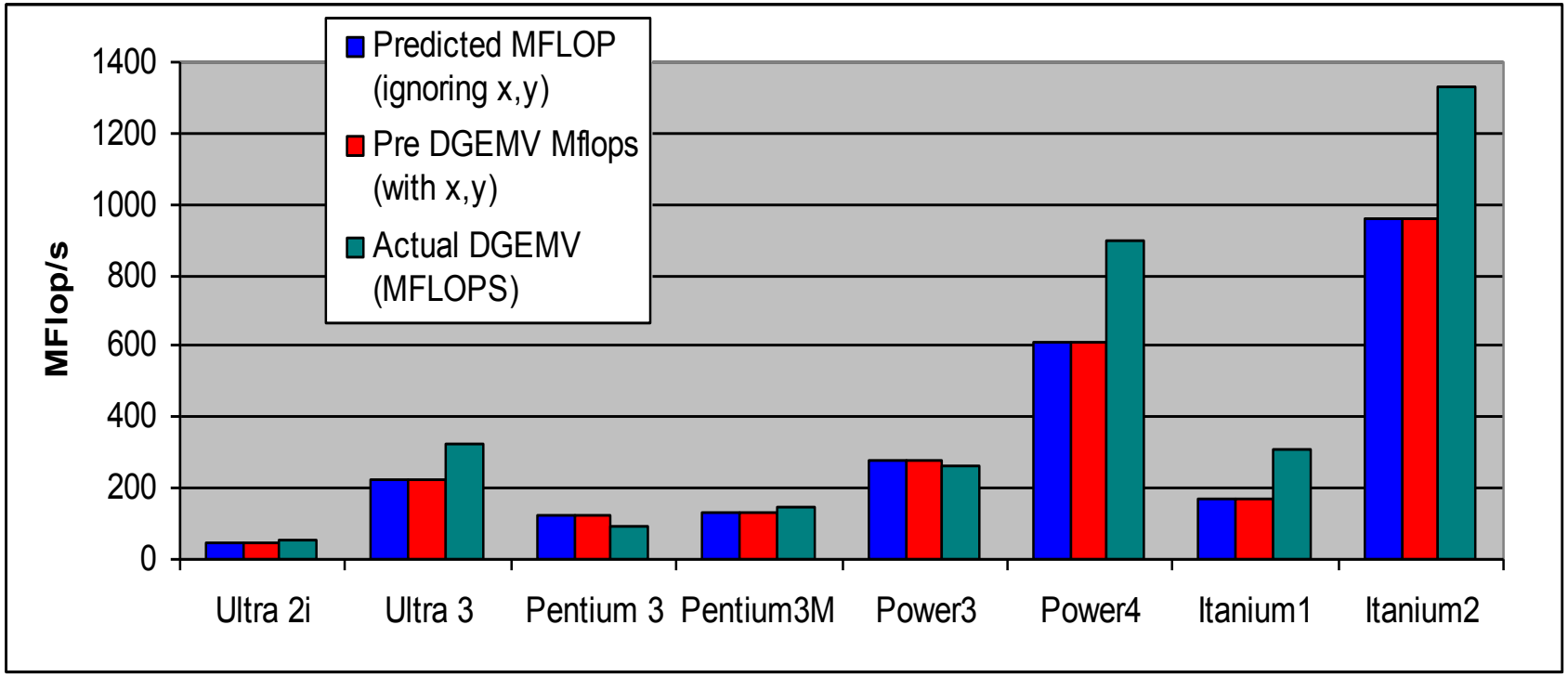
***machine balance* (q must be at least this for ½ peak speed)**

Simplifying Assumptions

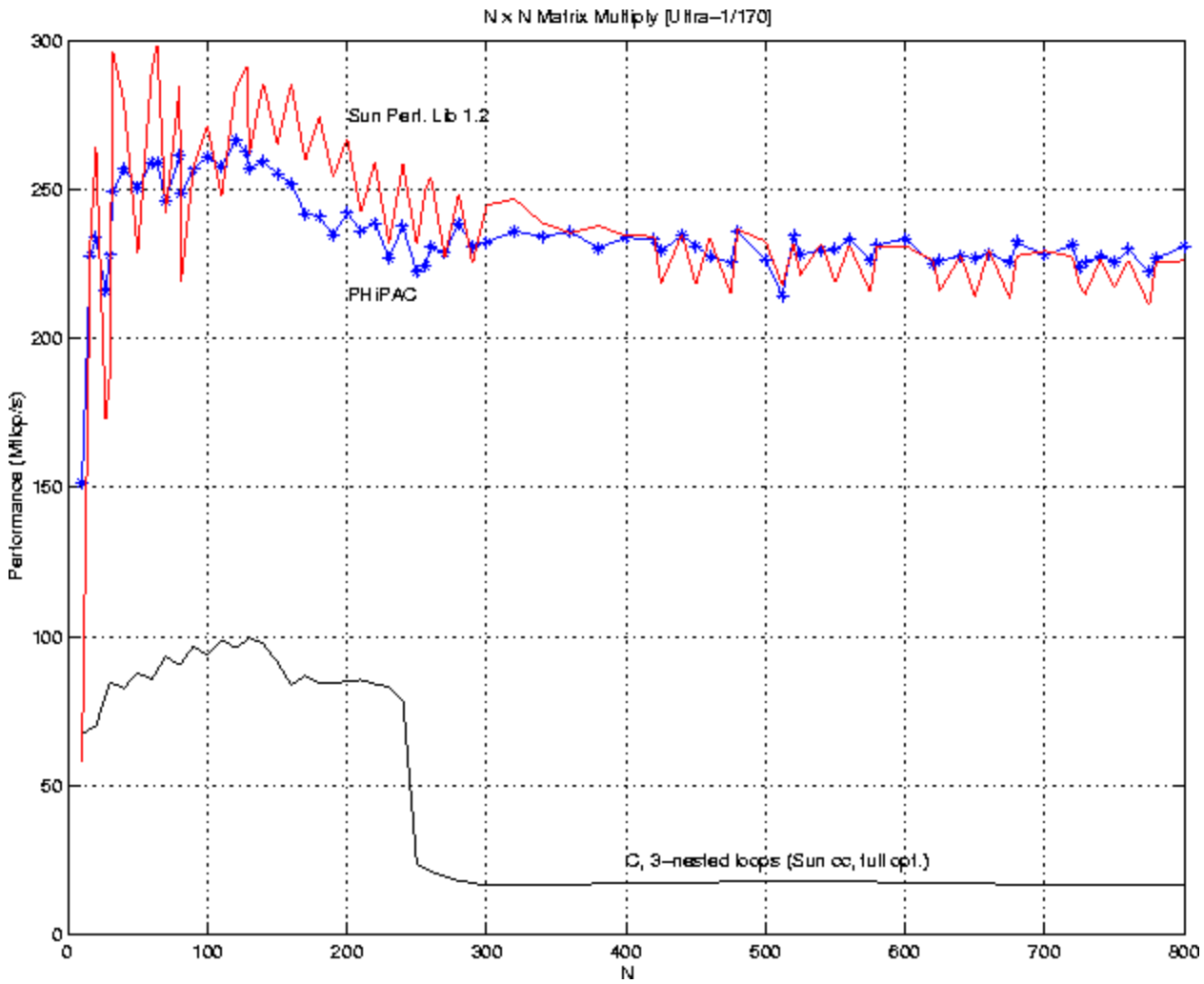
- What simplifying assumptions did we make in this analysis?
  - Ignored parallelism in processor between memory and arithmetic within the processor
    - Sometimes drop arithmetic term in this type of analysis
  - Assumed fast memory was large enough to hold three vectors
    - Reasonable if we are talking about any level of cache
    - Not if we are talking about registers (~32 words)
  - Assumed the cost of a fast memory access is 0
    - Reasonable if we are talking about registers
    - Not necessarily if we are talking about cache (1-2 cycles for L1)
  - Memory latency is constant

Validating the Model

- How well does the model predict actual performance?
  - Actual DGEMV: Most highly optimized code for the platform
- Model sufficient to compare across machines
- But under-predicting later ones due to latency estimate

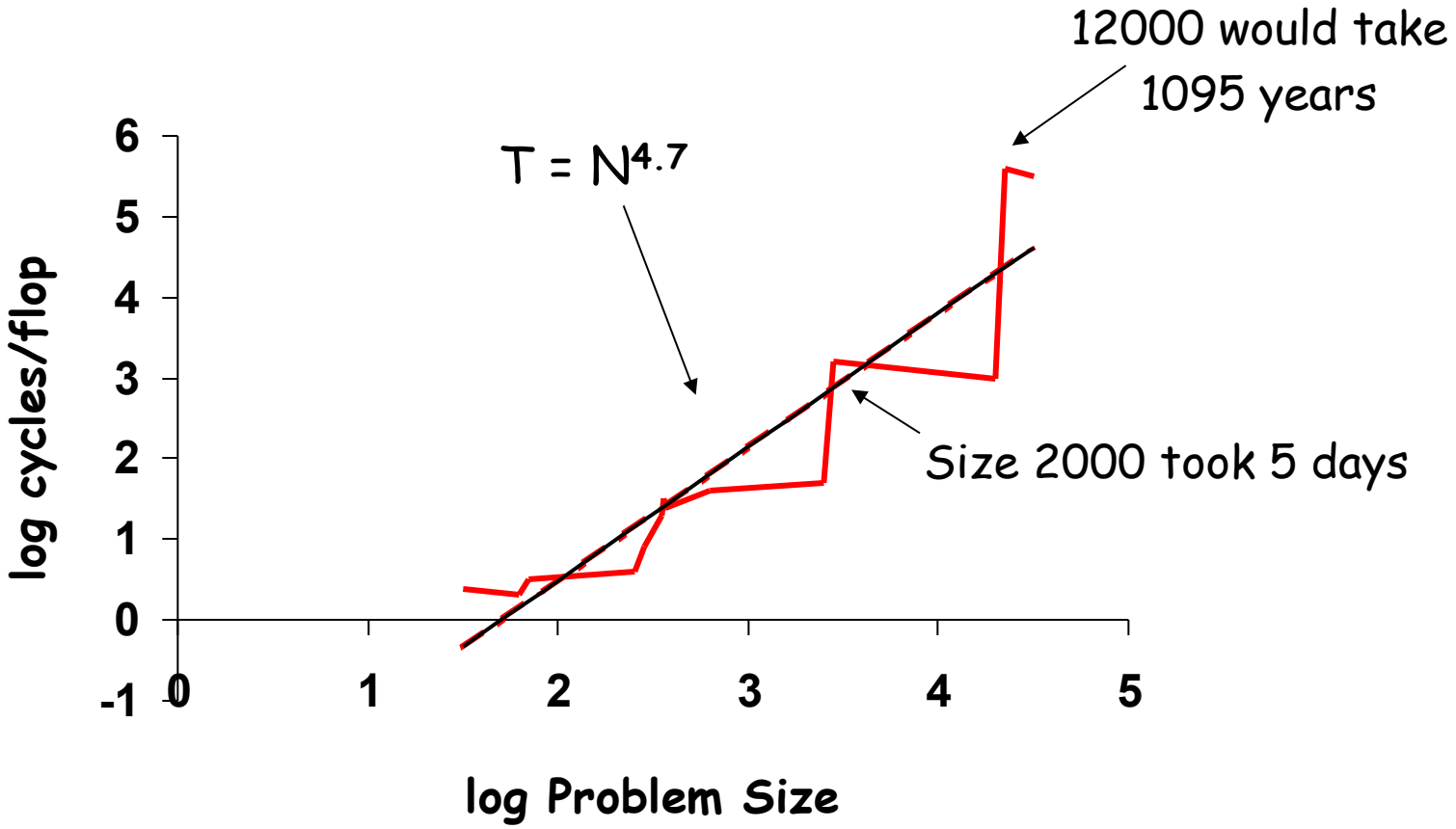


Matrix-multiply, optimized several ways



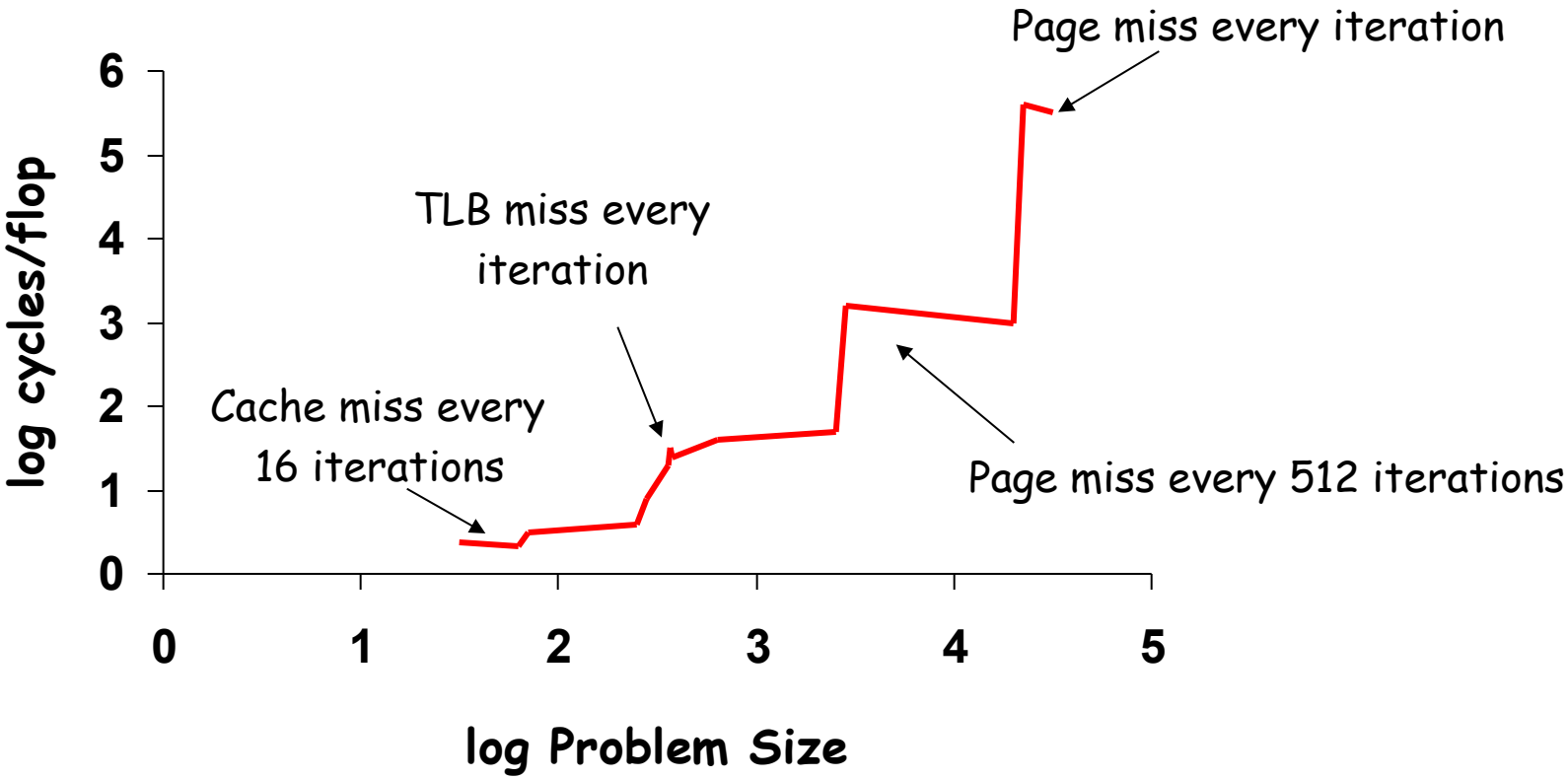
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

Naïve Matrix Multiply on RS/6000



O(N³) performance would have constant cycles/flop
Performance looks like O(N^4.7)

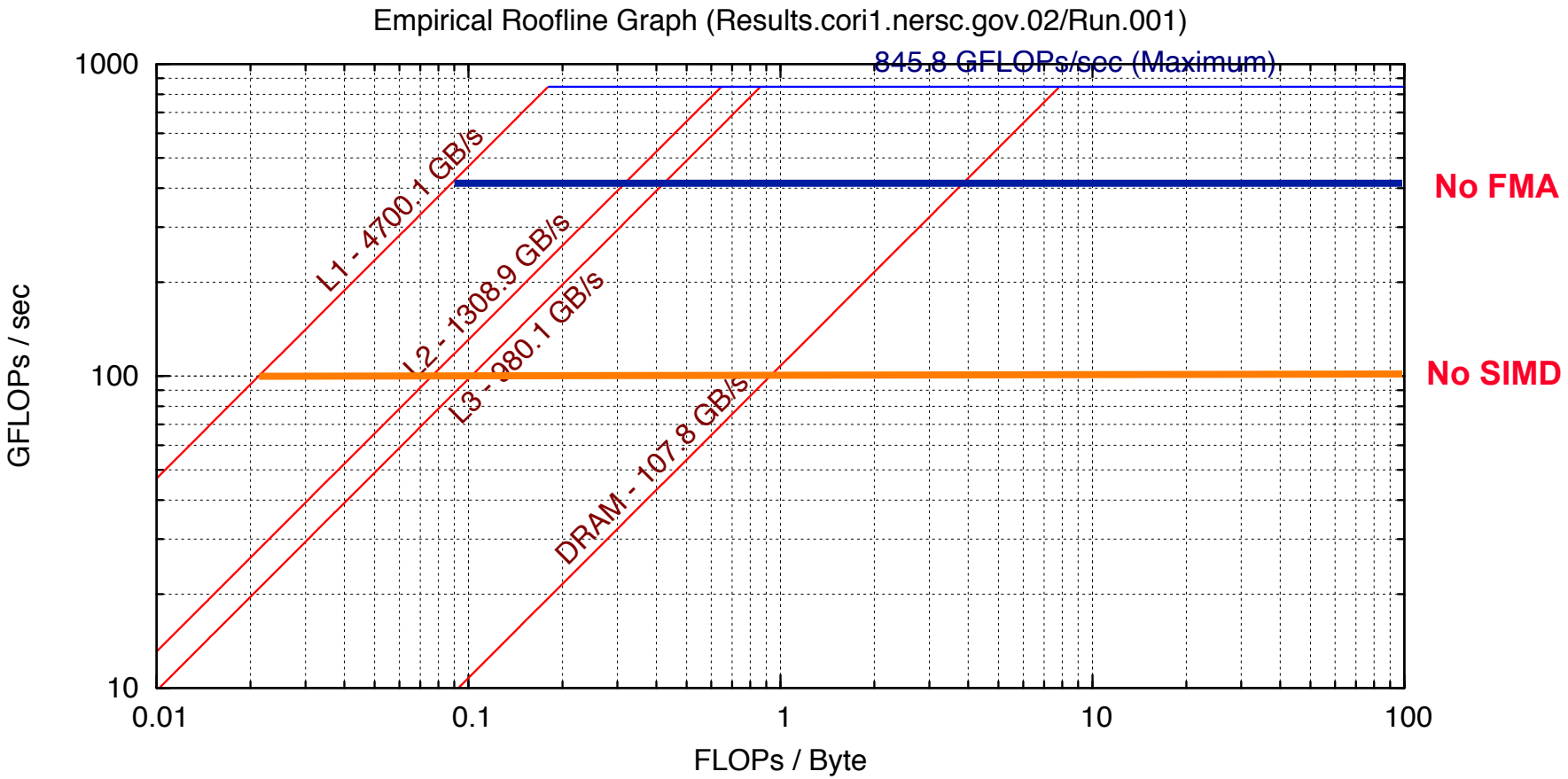
Naïve Matrix Multiply on RS/6000



Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case Study: Matrix MUltiplications
- Roofline Model
  - A simple model that allows us to understand algorithmic tradeoffs

Roofline Model (Williams, et al. 2009)



Summary

- Details of machine are important for performance
  - Processor and memory system
  - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
  - Pipelining, SIMD, etc
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby that recently used
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Need to rearrange code/data to improve locality