# CS267 Lecture 2

# Single Processor Machines: Memory Hierarchies and Processor Features

# Case Study: Tuning Matrix Multiply

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spr2018/

# Motivation

- Parallel computing is about performance
  - Distributed computing is a different thing
- Most applications run at < 10% of the "peak" performance
  - Peak is the maximum the hardware can physically execute
- Much of the performance is lost on a single processor
  - The code running on one processor often runs at only 10-20% of the processor peak
- Most of that loss is in the memory system
  - Moving data takes much longer than arithmetic and logic
- 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

# Possible conclusions to draw from today's lecture

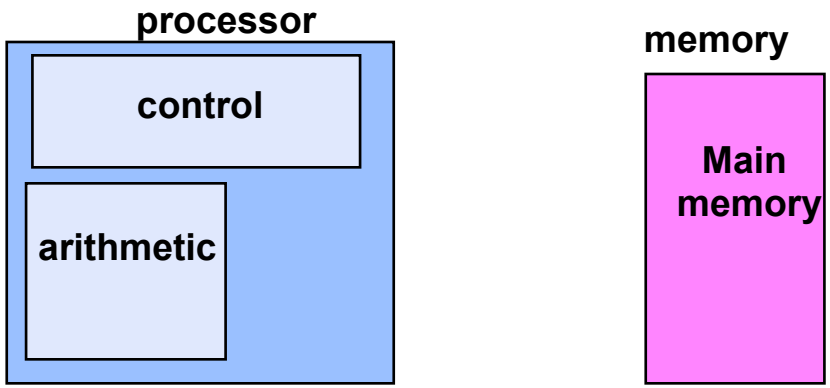
- "Computer architectures are fascinating, and I really want to understand why apparently simple programs can behave in such complex ways!"
- "I want to learn how to design algorithms that run really fast no matter how complicated the underlying computer architecture."
- "I hope that most of the time I can libraries or other software that hides all these details from me so I don't have to worry about them!"
- "I wish the compiler would handle everything for me."
- "I would like to write a compiler that would handle all of these details."
- I want to understand how Meltdown/Spectre work

# Outline

- Costs in modern processors
  - Idealized models and actual costs
- Memory hierarchies
- Parallelism in a single processor
- Case study: Matrix multiplication

# Idealized Uniprocessor Model

- Processor names variables:
  - Integers, floats, pointers, arrays, structures, etc.
- Processor performs operations on those variables:
  - Arithmetic, logical operations, etc.
- Processor controls the order, as specified by program
  - Branches (if), loops, function calls, etc.

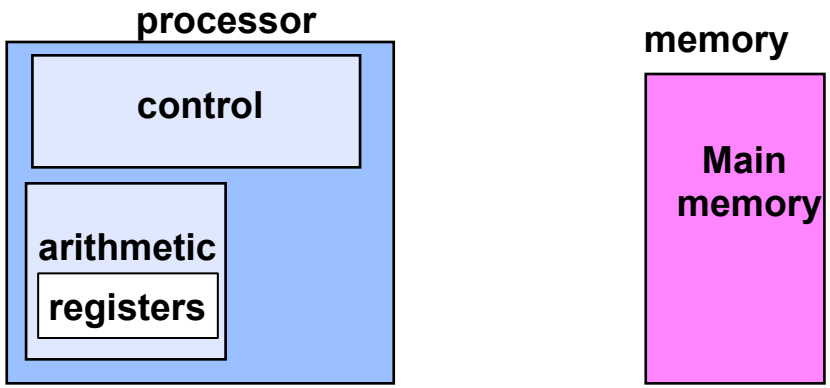



- *Idealized Cost*
  - Each operation has roughly the same cost

    add, multiply, etc.

# Slightly Less Idealized Uniprocessor Model

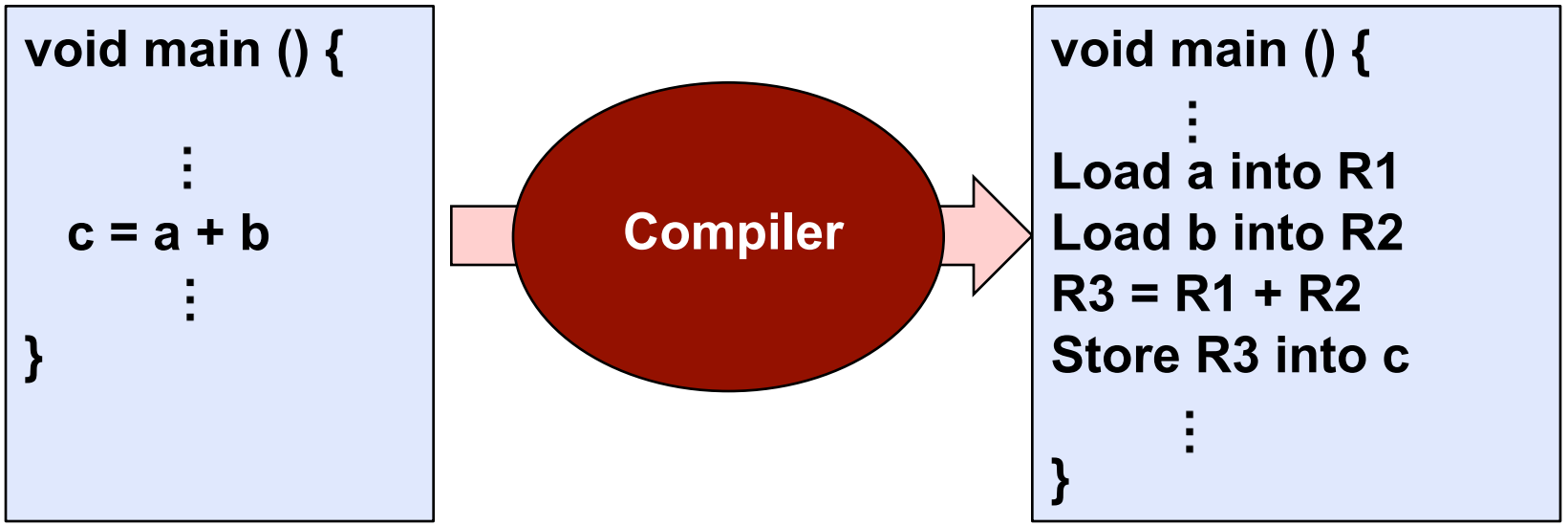
- Processor names variables:
  - Integers, floats, pointers, arrays, structures, etc.
  - These are really words, e.g., 64-bit doubles, 32-bit ints, bytes, etc.
- Processor performs operations on those variables:
  - Arithmetic, logical operations, etc.
  - Only performs these operations on values in registers
- Processor controls the order, as specified by program
  - Branches (if), loops, function calls, etc.




- *Idealized Cost*
  - Each operation has roughly the same cost

    add, multiply, etc.

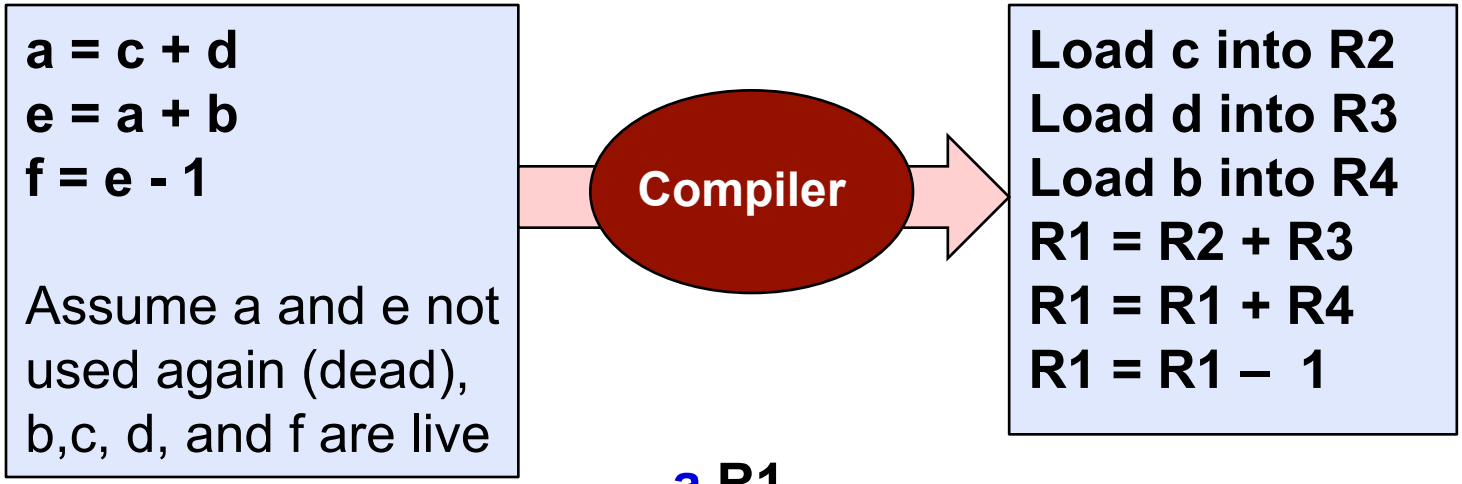
# Compilers and assembly code

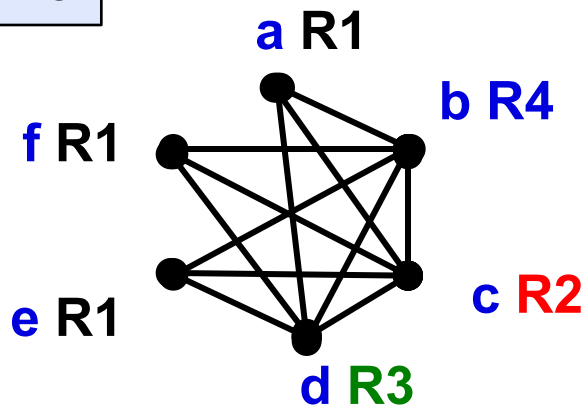



- Compilers for languages like C/C++ and Fortran:
  - Check that the program is legal
  - Translate into assembly code
  - Optimizes the generated code

# Compilers Manage Memory and Registers

- **Compiler performs "register allocation" to decide when to load/store and when to reuse**




*Register allocation in first Fortran compiler in 1950s, graph coloring in 1980. JITs may use something cheaper*

# Compiler Optimizes Code

- **Compiler performs a number of optimizations:**
  - **Unrolls loops (because control isn't free)**
  - **Fuses loops (merge two together)**
  - **Interchanges loops (reorder)**
  - **Eliminates dead code (the branch never taken)**
  - **Reorders instructions to improve register reuse and more**
  - **Strength reduction (turns expensive instruction, e.g., multiply by 2, into cheaper one, shift left)**

- **The compiler managed operations (load/store) that are important to performance but we don't "see" in code.**

- **Why is this your problem?**
  - **Because sometimes it does the best thing possible**
  - **But other times it does not…**

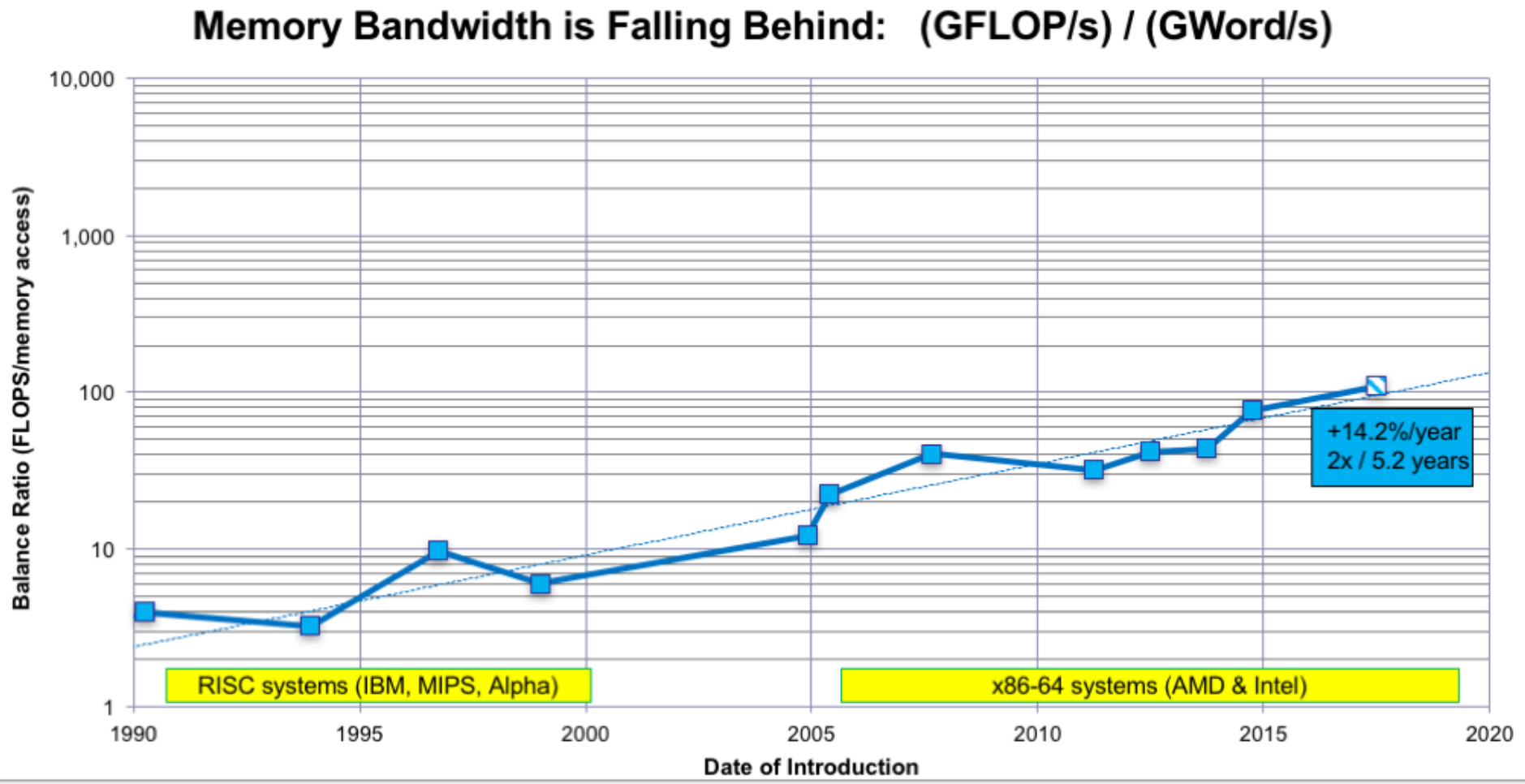
# Outline

- 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

# Outline

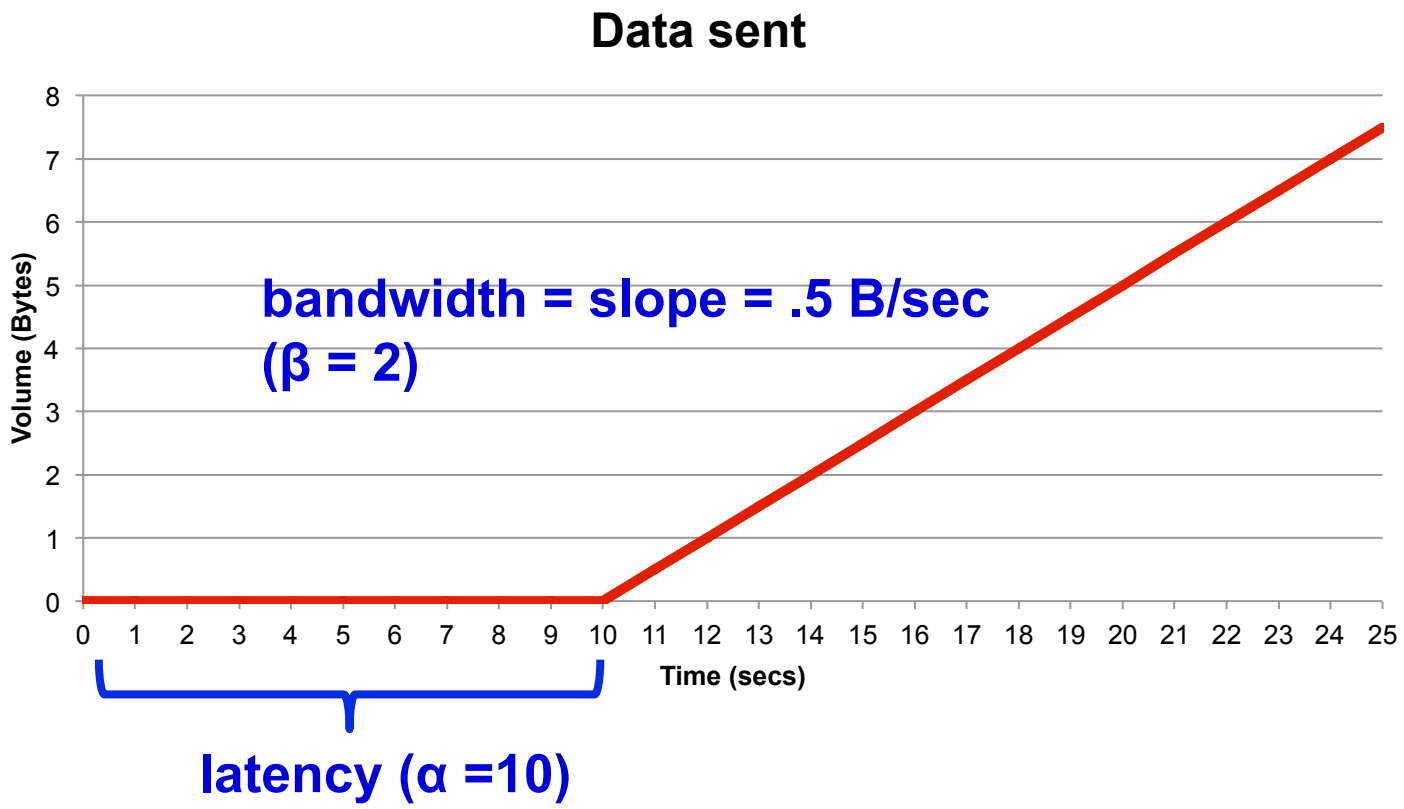
- Costs in modern processors
- Memory hierarchies
  - Temporal and spatial locality
  - Basics of caches
  - Use of microbenchmarks to characterize performance
- Parallelism in a single processor
- Case study: Matrix multiplication

# Memory Bandwidth Gap

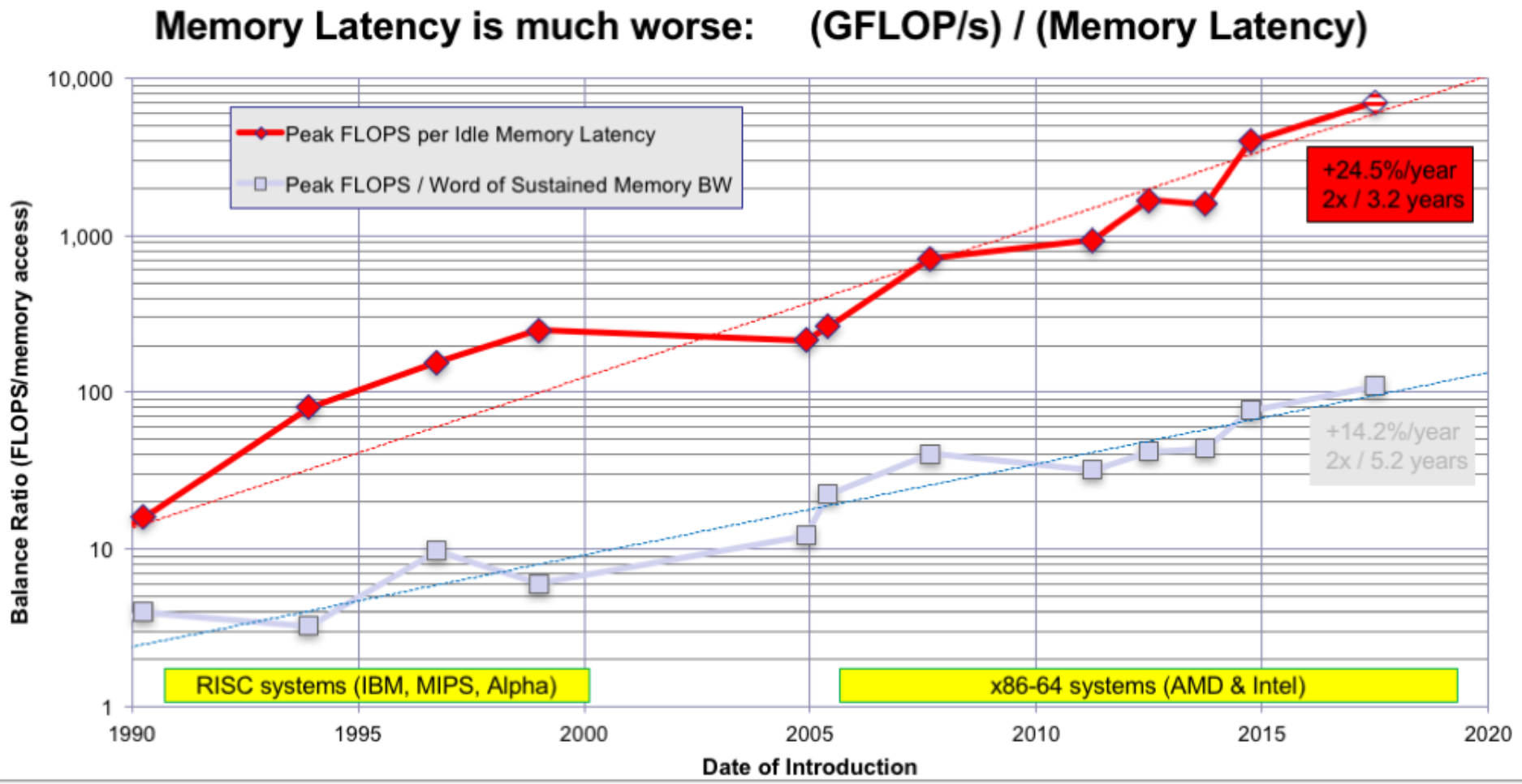


# Latency vs Bandwidth

- Previous graph show bandwidth gap
- Many applications are limited not by memory bandwidth, but by latency

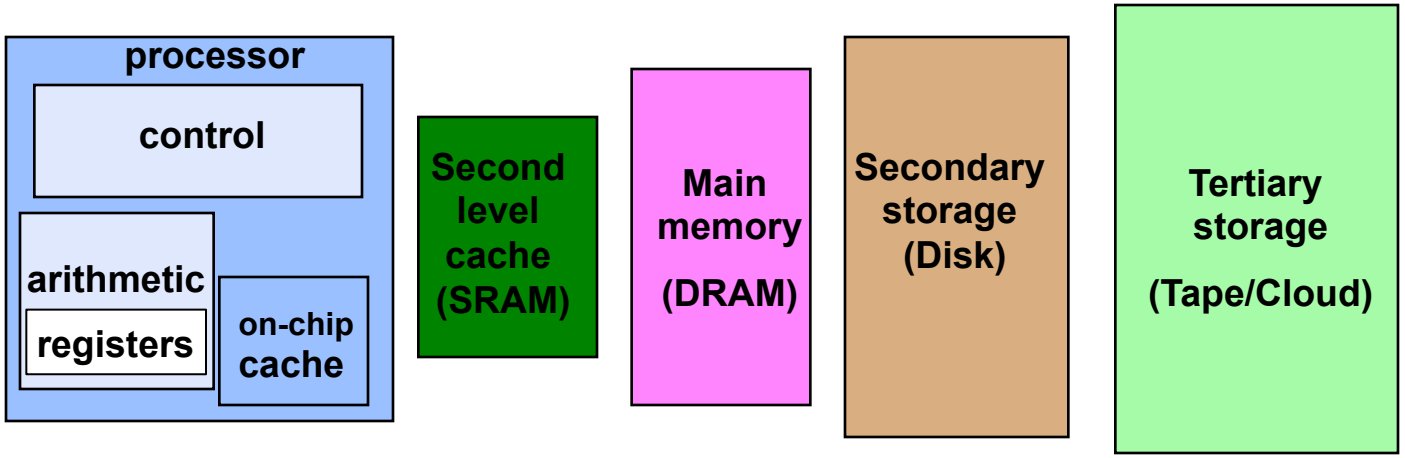


# Memory Latency Gap is Worse



# Memory Hierarchy

- Most programs have a high degree of locality
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy use this to improve *average case*




| | | | | | |
|---|---|---|---|---|---|
| Speed | 1ns | 10ns | 100ns | 10ms | 10sec |
| Size | KB | MB | GB | TB | PB |

# 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**
    - **Up to $n \leq 16$ words with addresses xxxxx1101 can be stored at cache location 1101 (so cache can store 16n words)**

# 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**
  - **Intel Haswell (Cori Ph1) uses a Level 4 cache as "victim cache"**
- There are other levels of the memory hierarchy
  - **Register, pages (TLB, virtual memory), …**
  - **And it isn't always a hierarchy**

# Approaches to Handling Memory Latency

- Eliminate memory operations by saving values in small, fast memory (cache or registers) and reusing them (bandwidth filtering)
  - **need temporal locality in program**
- Take advantage of better bandwidth by getting a chunk of memory into cache (or vector registers) and using whole chunk
  - **need spatial locality in program**
- Take advantage of better bandwidth by allowing processor to issue multiple reads or writes with a single instruction
  - **vector operations require access set of locations (typically neighboring), requires they are independent**
- Take advantage of better bandwidth by allowing processor to issue reads/writes in parallel with other reads/writes/operations
  - **prefetching issues read hint**
  - **delayed writes (write buffering) stages writes for later operation**
  - **both require that nothing dependent is happening (parallelism)**

**concurrency**

# How much concurrency do you need?

- To run at bandwidth speeds rather than latency
- Little's Law from queuing theory says:

  **concurrency = latency * bandwidth**

- In my earlier example with

  latency = 10 sec

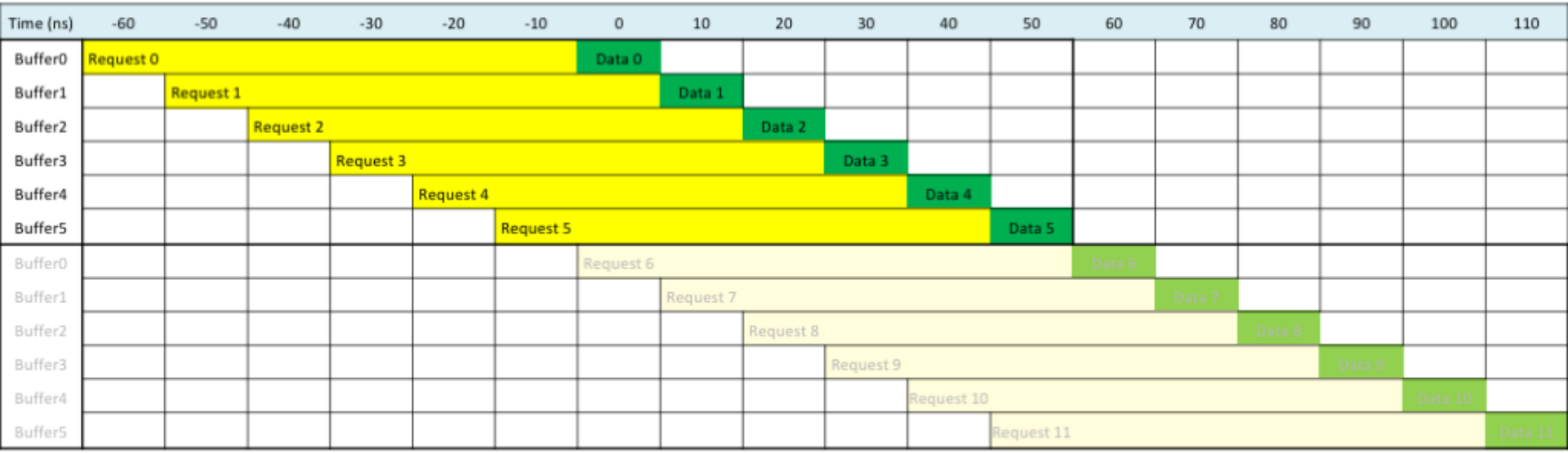
  bandwidth = 2 Bytes/sec

- Requires 20 bytes in flight concurrently to reach bandwidth speeds
- That means finding 20 independent things to issue

# More realistic example

## Little's Law: illustration for 2005-era Opteron processor

60 ns latency, 6.4 GB/s (=10ns per 64B cache line)



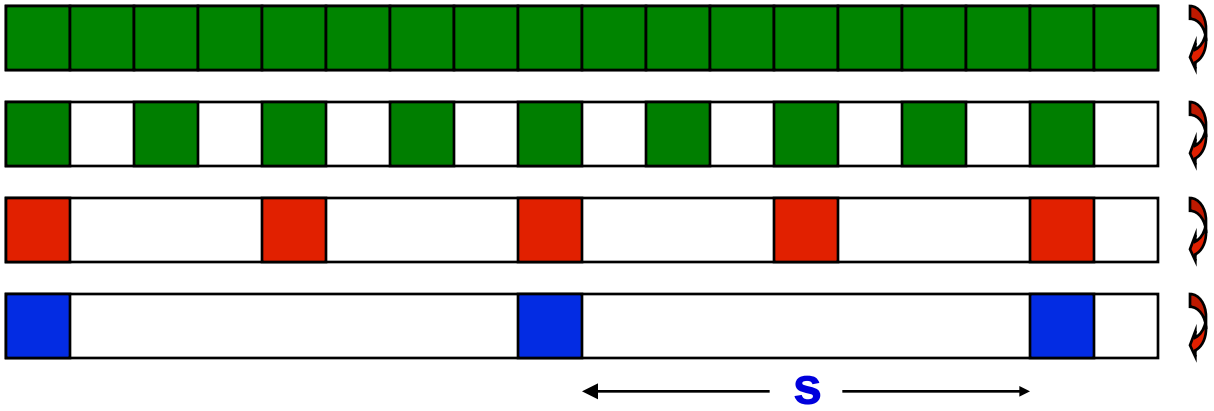

- 60 ns * 6.4 GB/s = 384 Bytes = 6 cache lines
- To keep the pipeline full, there must always be 6 cache lines "in flight"
- Each request must be launched at least 60 ns before the data is needed

# Stream benchmark for measuring bandwidth

- Stream benchmark (also due to McCalpin
- Four kernels, all for i = 1 to N:
  - `Copy:  C[i] = A[i];`
  - `Scale:  B[i]  = scalar * C[i];`
  - `Add:  C[i] = A[i] + B[i];`
  - `Triad:  A[i] = B[i] + scalar * C[i];`
- N chose so that array is much larger than cache
- Repeated ~10 times (ignoring first) to “warm cache
- Min/Av/Max timing report

# Experimental Study of Memory (Membench)

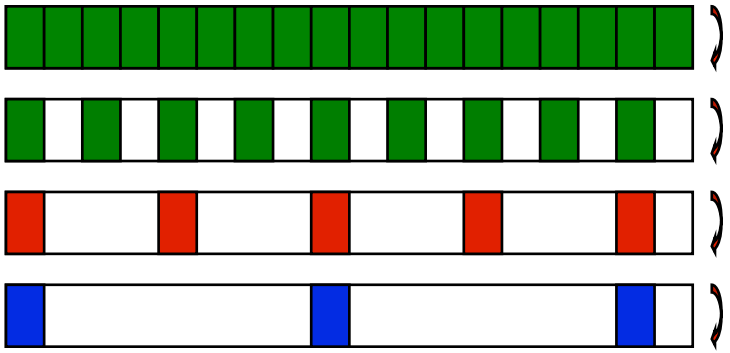
- Microbenchmark for memory system performance

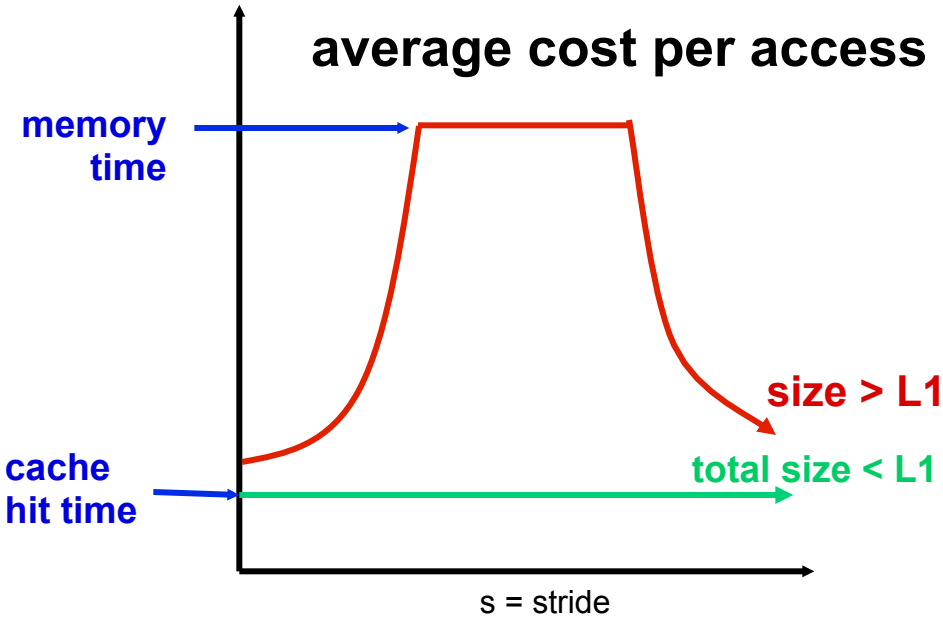



- **for array A of length L from 4KB to 8MB by 2x**
  **for stride s from 4 Bytes (1 word) to L/2 by 2x**
  **time the following loop**
  **(repeat many times and average)**
  **for i from 0 to L-1 by s**
  **load A[i] from memory (4 Bytes)**

**1 experiment**

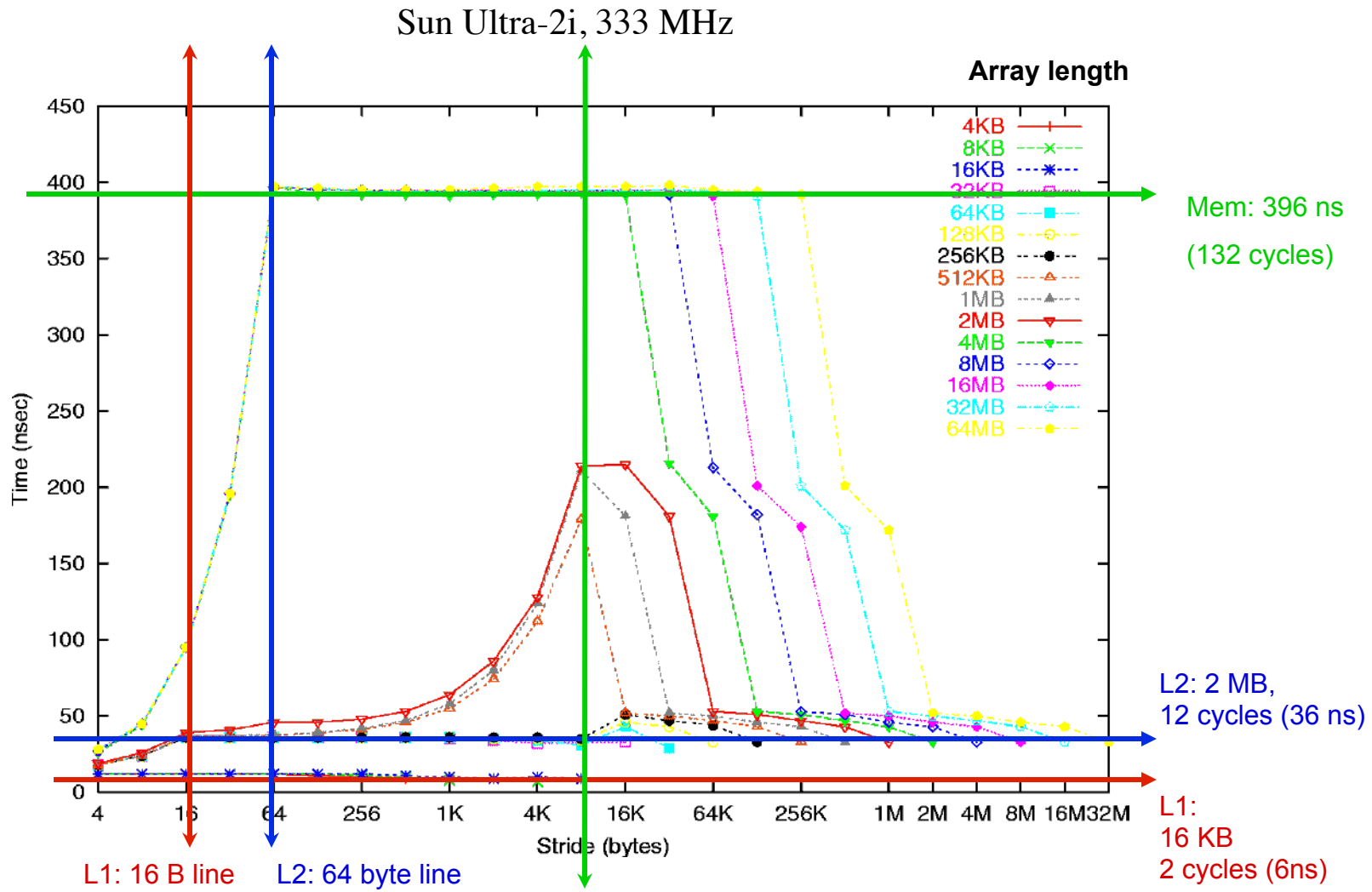
# Membench: What to Expect






- 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

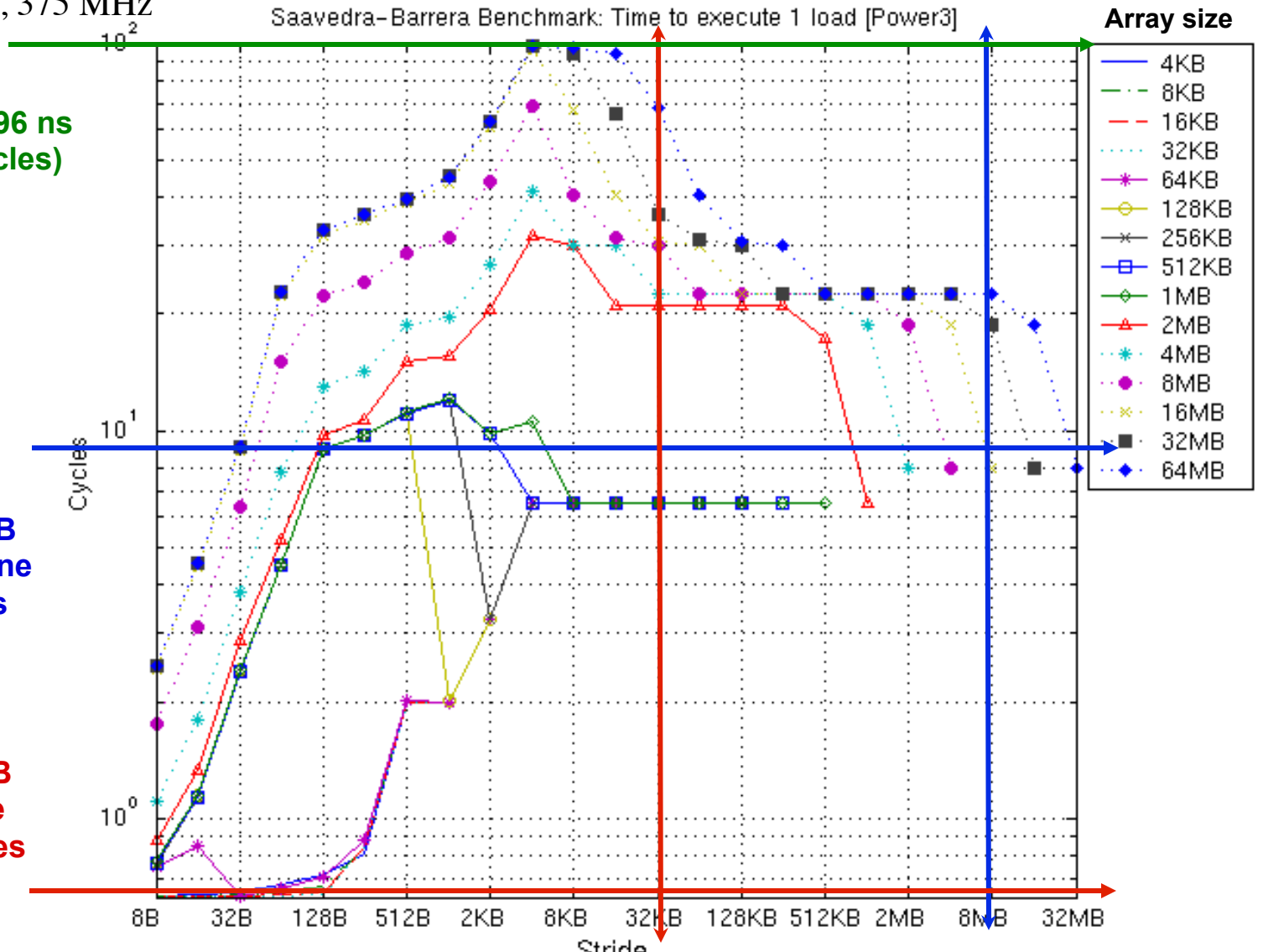
# Memory Hierarchy on a Sun Ultra-2i




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

# Memory Hierarchy on a Power3 (Seaborg)

Power3, 375 MHz

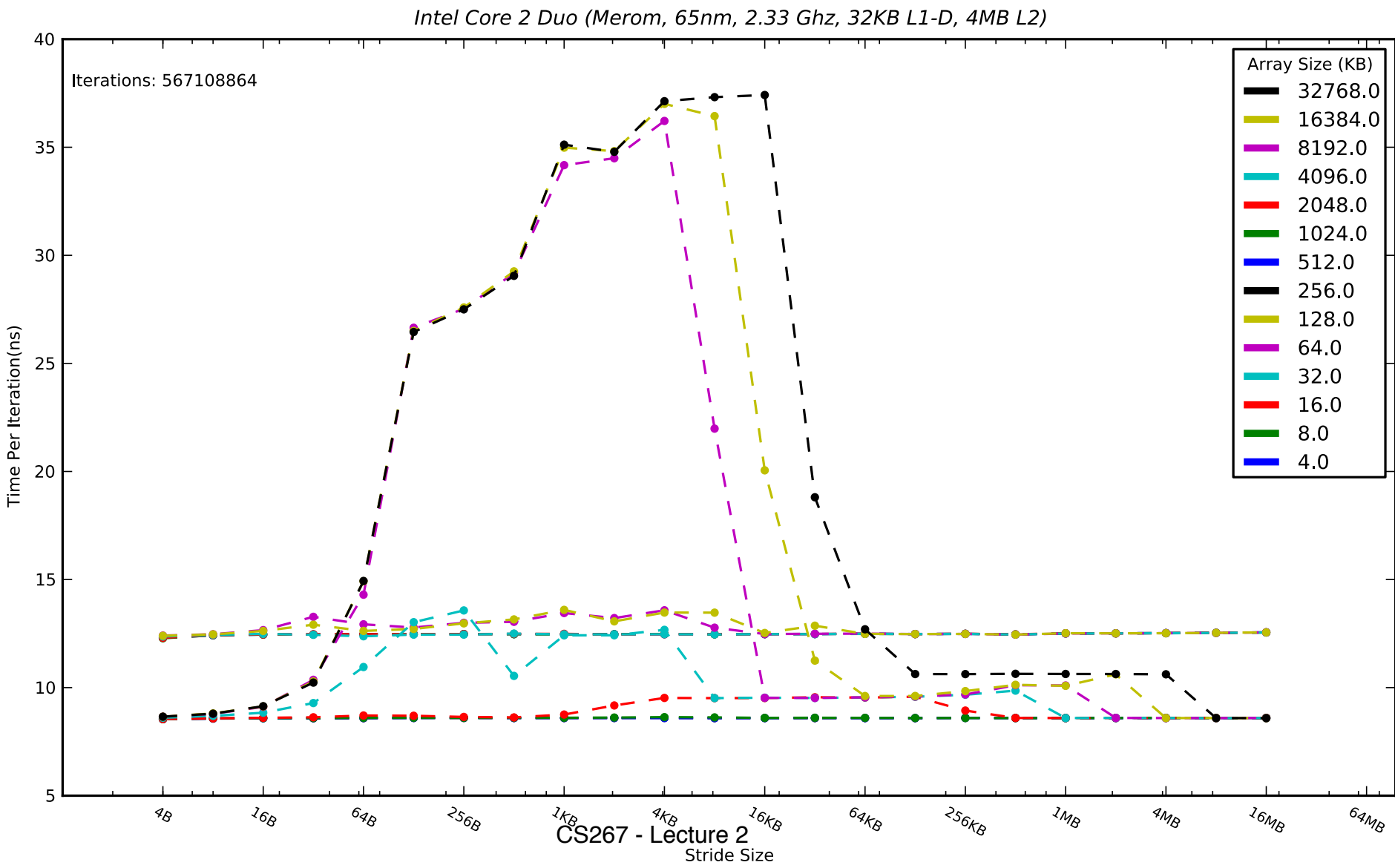



**Mem: 396 ns (132 cycles)**

**L2: 8 MB**
**128 B line**
**9 cycles**

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

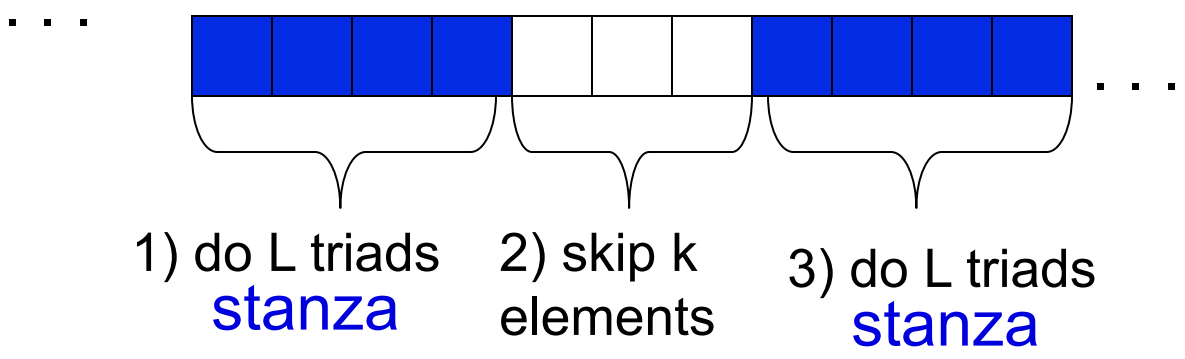
# Memory Hierarchy on an Intel Core 2 Duo



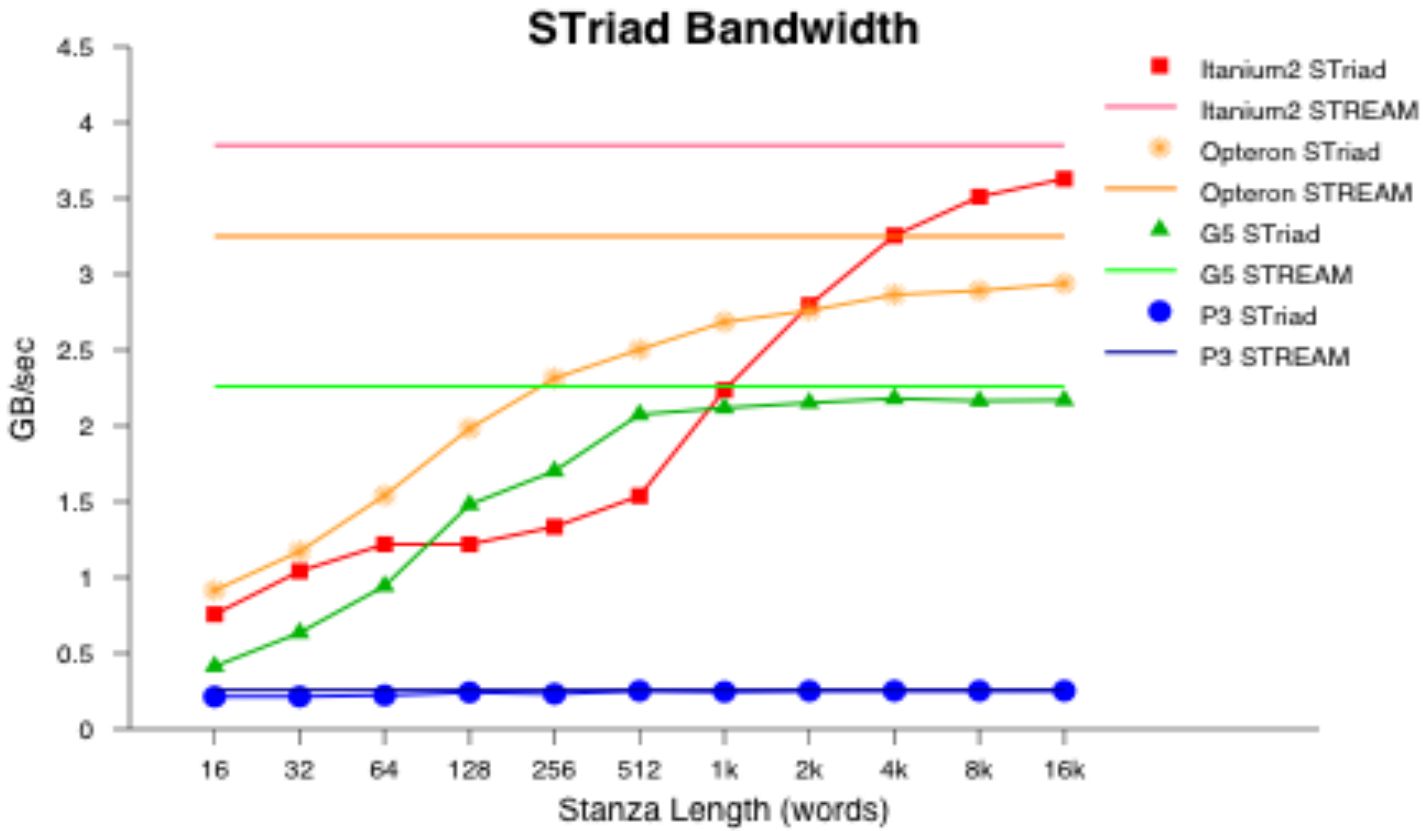
# Stanza Triad

- 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
```

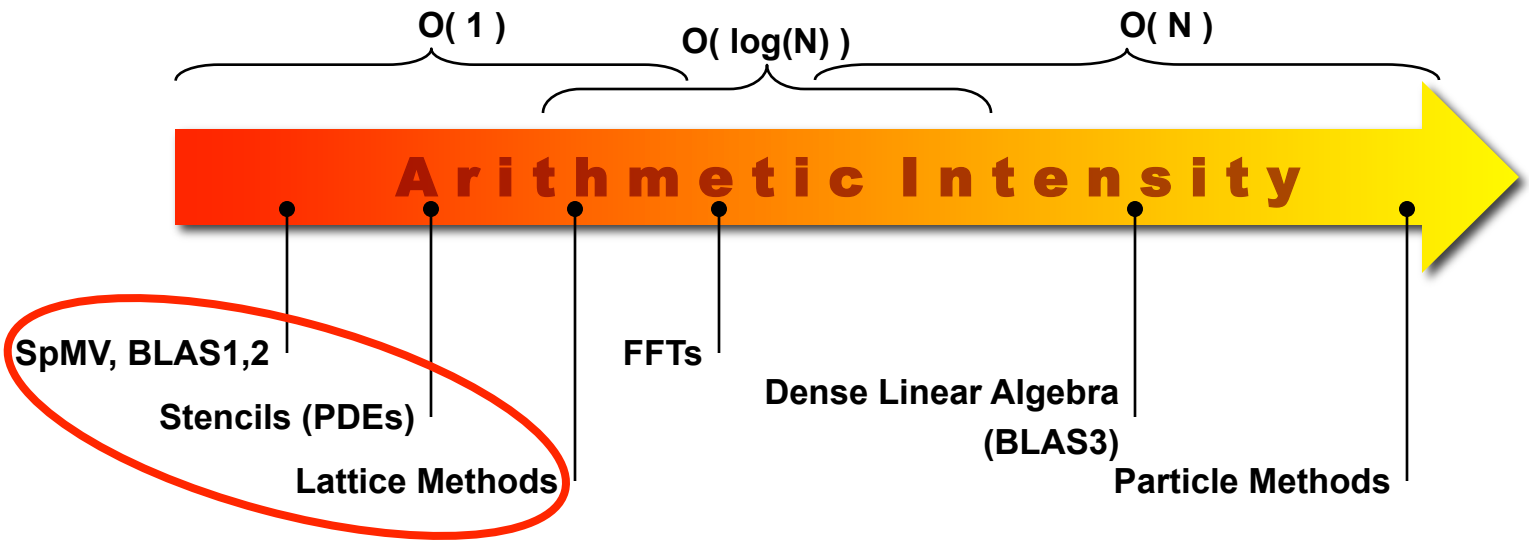


# Stanza Triad Results – Prefetch matters



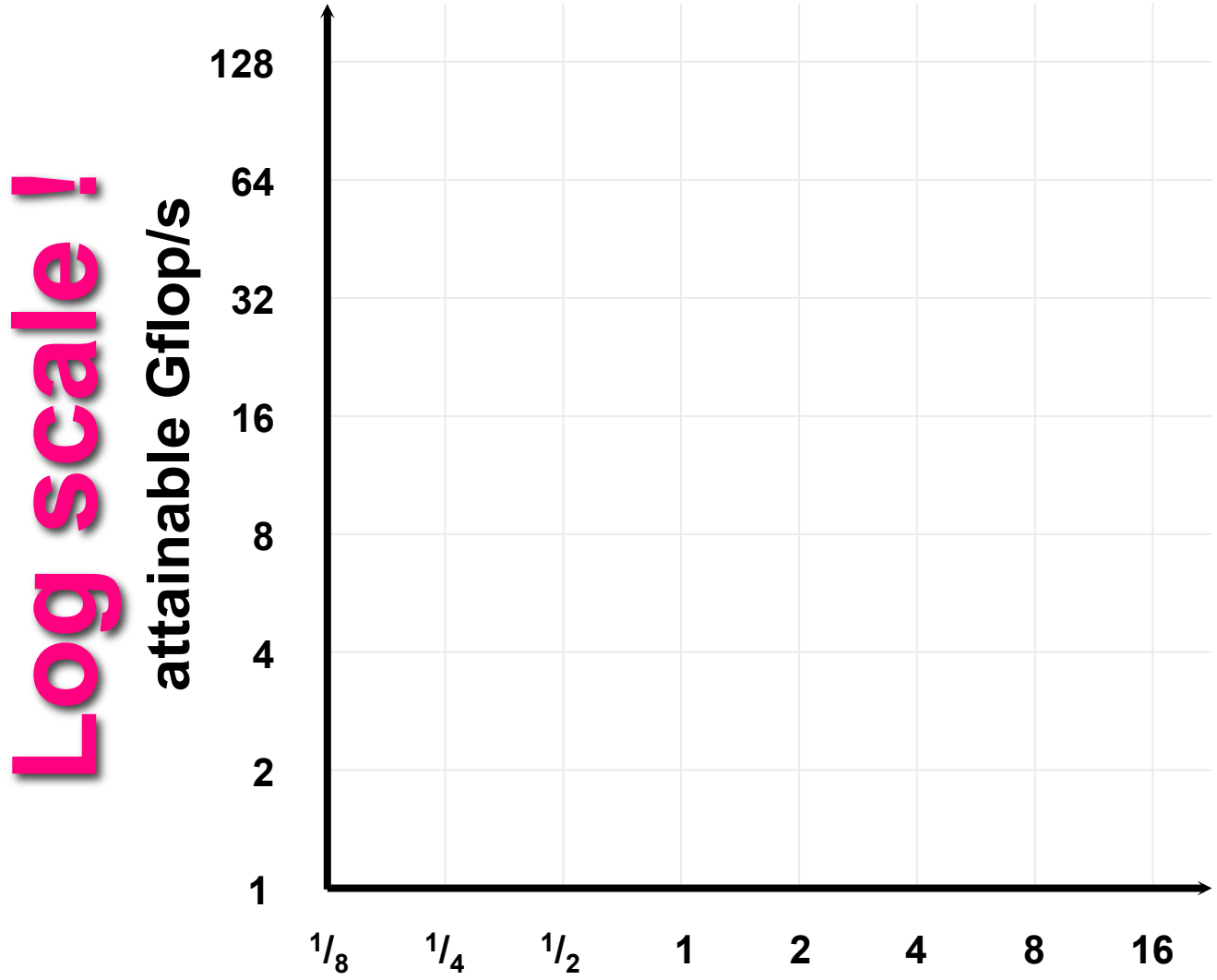

- This graph (x-axis) starts at a cache line size (>=16 Bytes)
- If cache locality was the only thing that mattered, we would expect
  - Flat lines equal to measured memory peak bandwidth (STREAM) as on Pentium3
- Prefetching gets the next cache line (pipelining) while using the current one
  - This does not "kick in" immediately, so performance depends on L
  - http://crd-legacy.lbl.gov/~oliker/papers/msp_2005.pdf

# Arithmetic Intensity



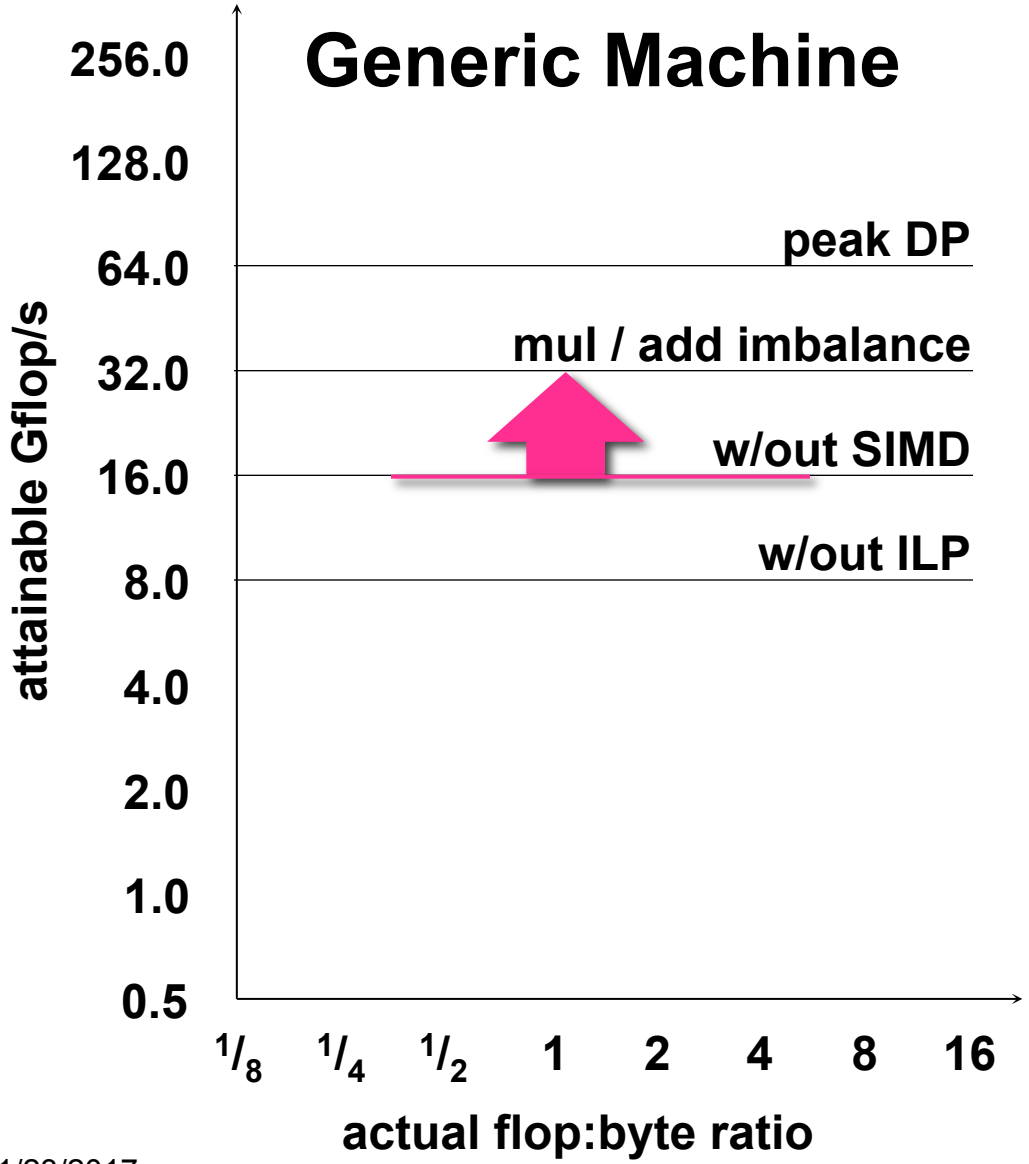

- Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes moved
- Can we hit the bandwidth or flop limit?
- The rest is all about having and being able utilize enough concurrency

# The Roofline Performance Model



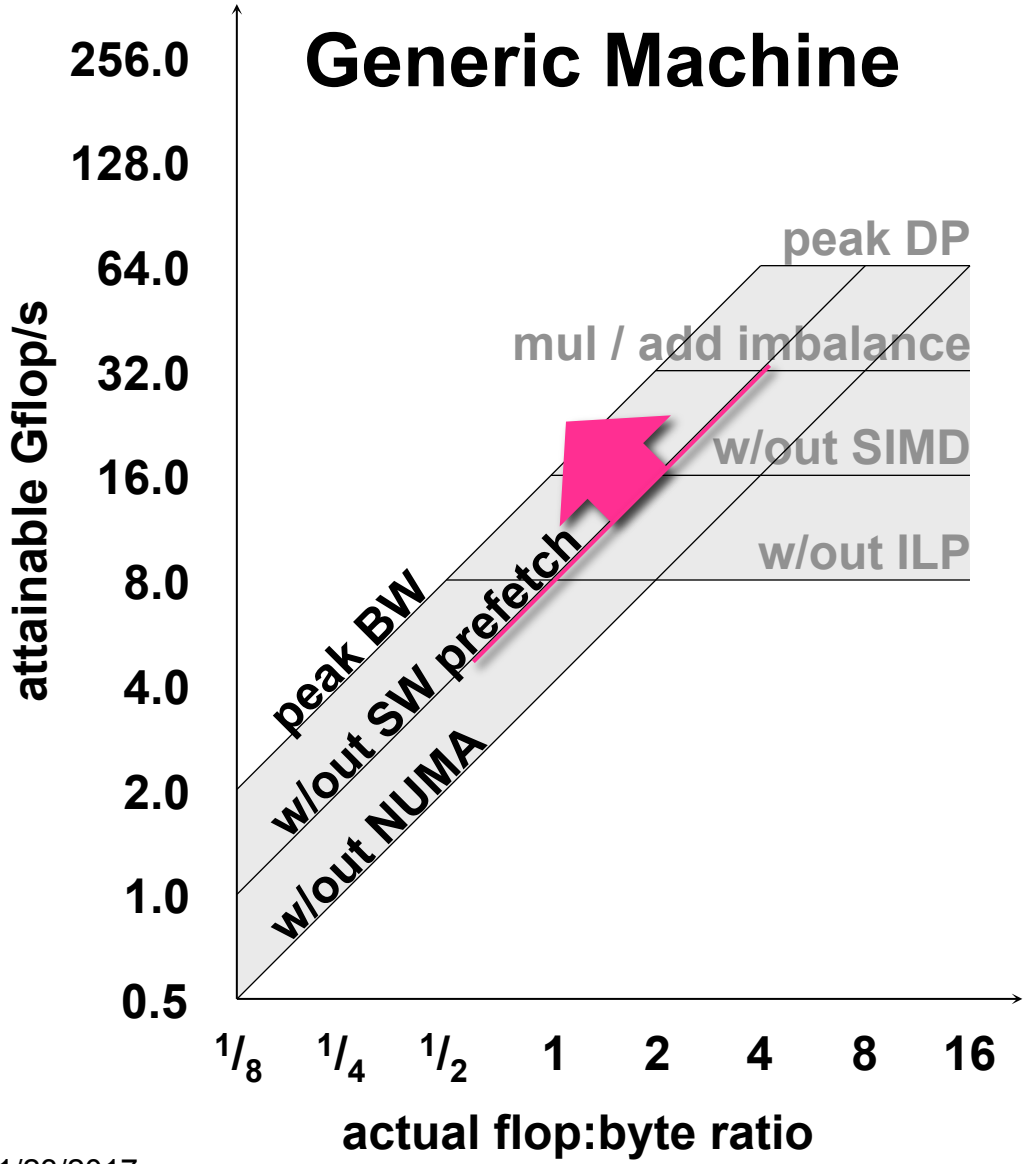

Sam Williams PhD, 2008

# The Roofline Performance Model



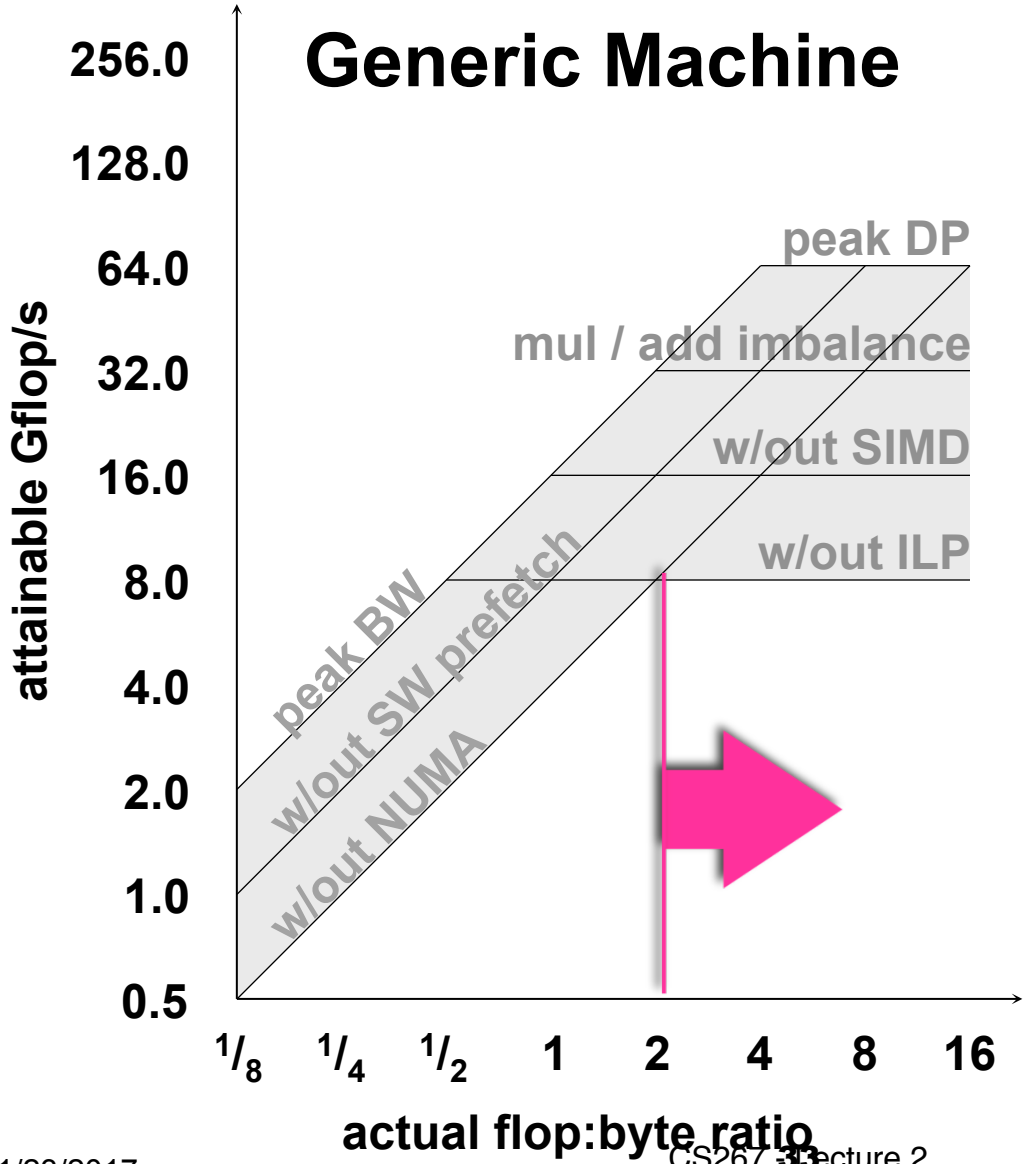

- The top of the roof is determined by peak computation rate (Double Precision floating point, DP for these algorithms)
- The instruction mix, lack of SIMD operations, ILP or failure to use other features of peak will lower attainable

# The Roofline Performance Model




- The sloped part of the roof is determined by peak DRAM bandwidth (STREAM)
- Lack of proper prefetch, ignoring NUMA, or other things will reduce attainable bandwidth

# The Roofline Performance Model




- Locations of posts in the building are determined by algorithmic intensity
- Will vary across algorithms and with bandwidth-reducing optimizations, such as better cache re-use (tiling), compression techniques

# 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
    - True on sequential or parallel processor
  - 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

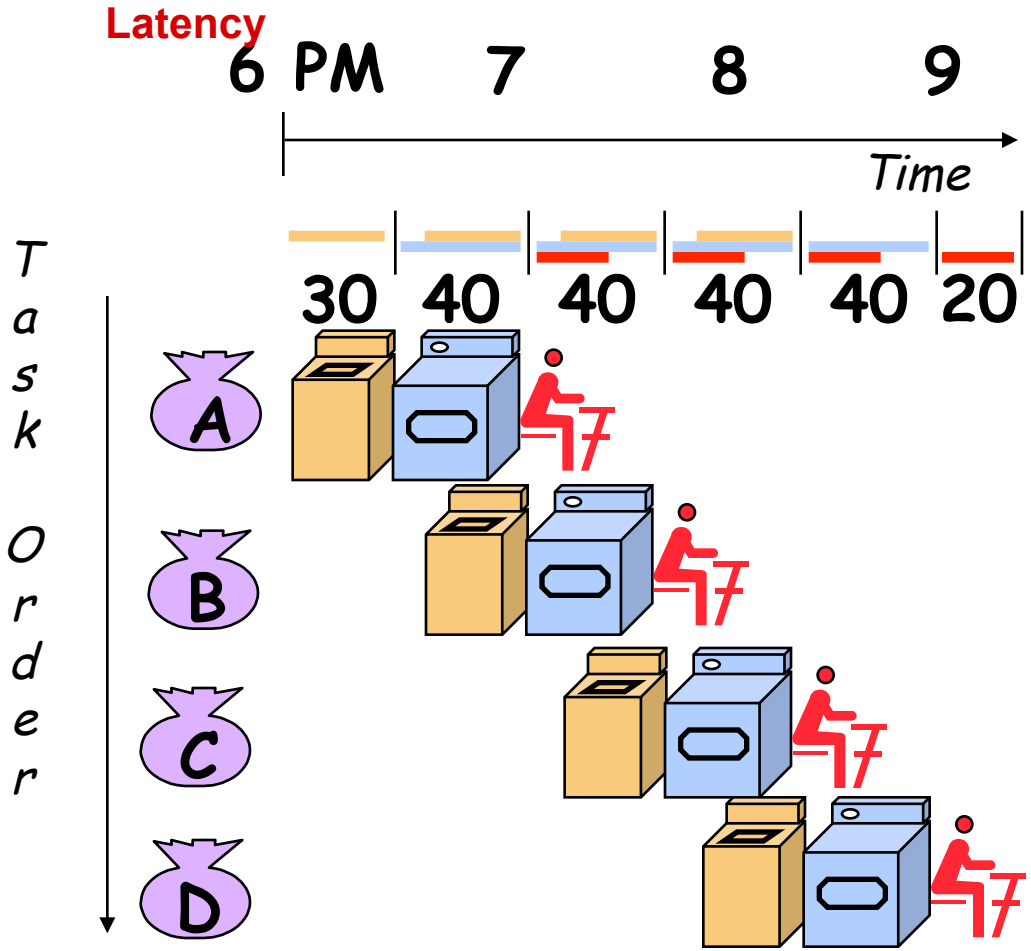
# Outline

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

# What is Pipelining?

**Dave Patterson's Laundry example: 4 people doing laundry**

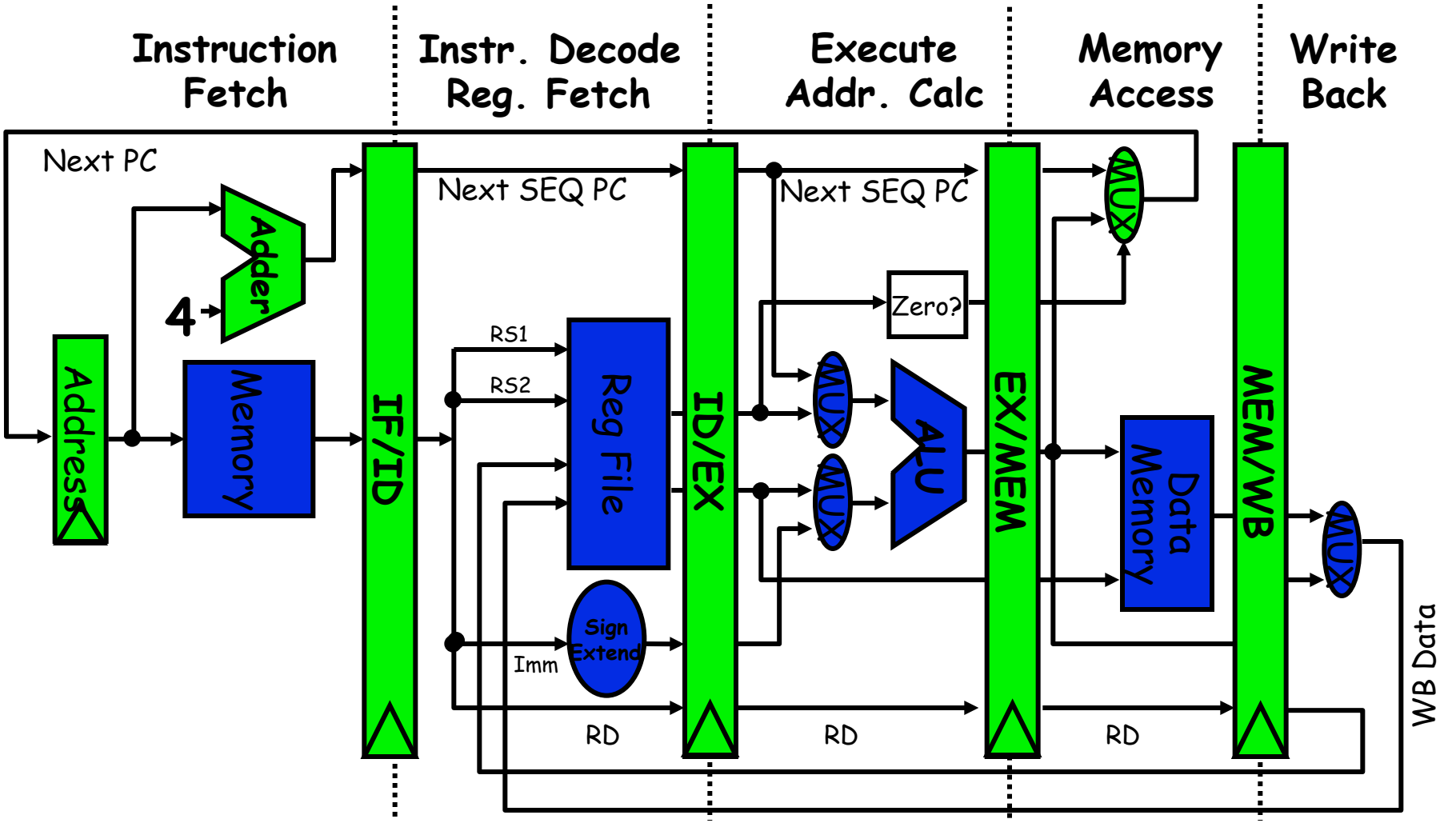
**wash (30 min) + dry (40 min) + fold (20 min) = 90 min**




- 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 doesn't change latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Speedup <= # of 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

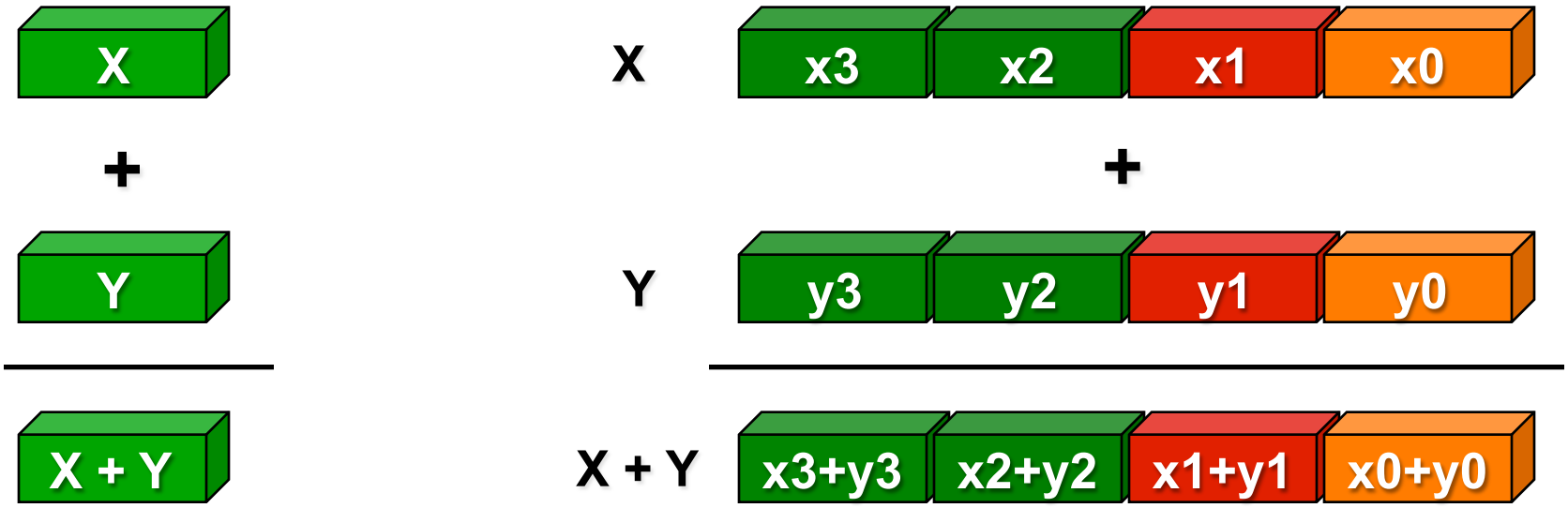
# SIMD: Single Instruction, Multiple Data

Scalar processing

- traditional mode
- one operation produces one result

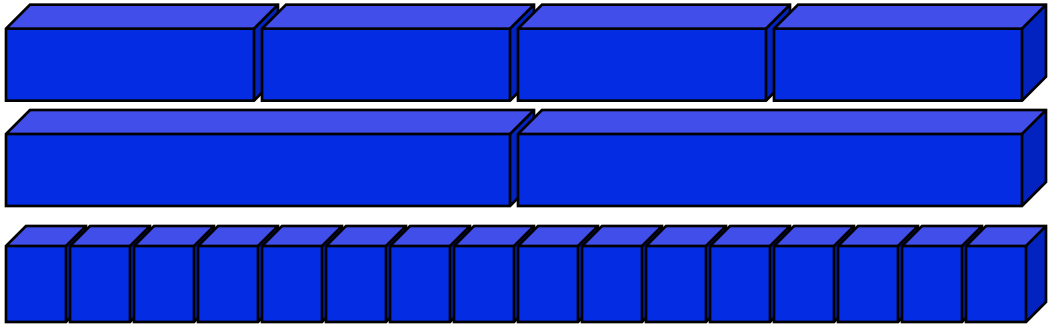
SIMD processing: vectors

- Sandy Bridge: AVX (256 bit)
- Haswell: AVX2 (256 bit w/ FMA)
- KNL: AVX-512 (512 bit)



# 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

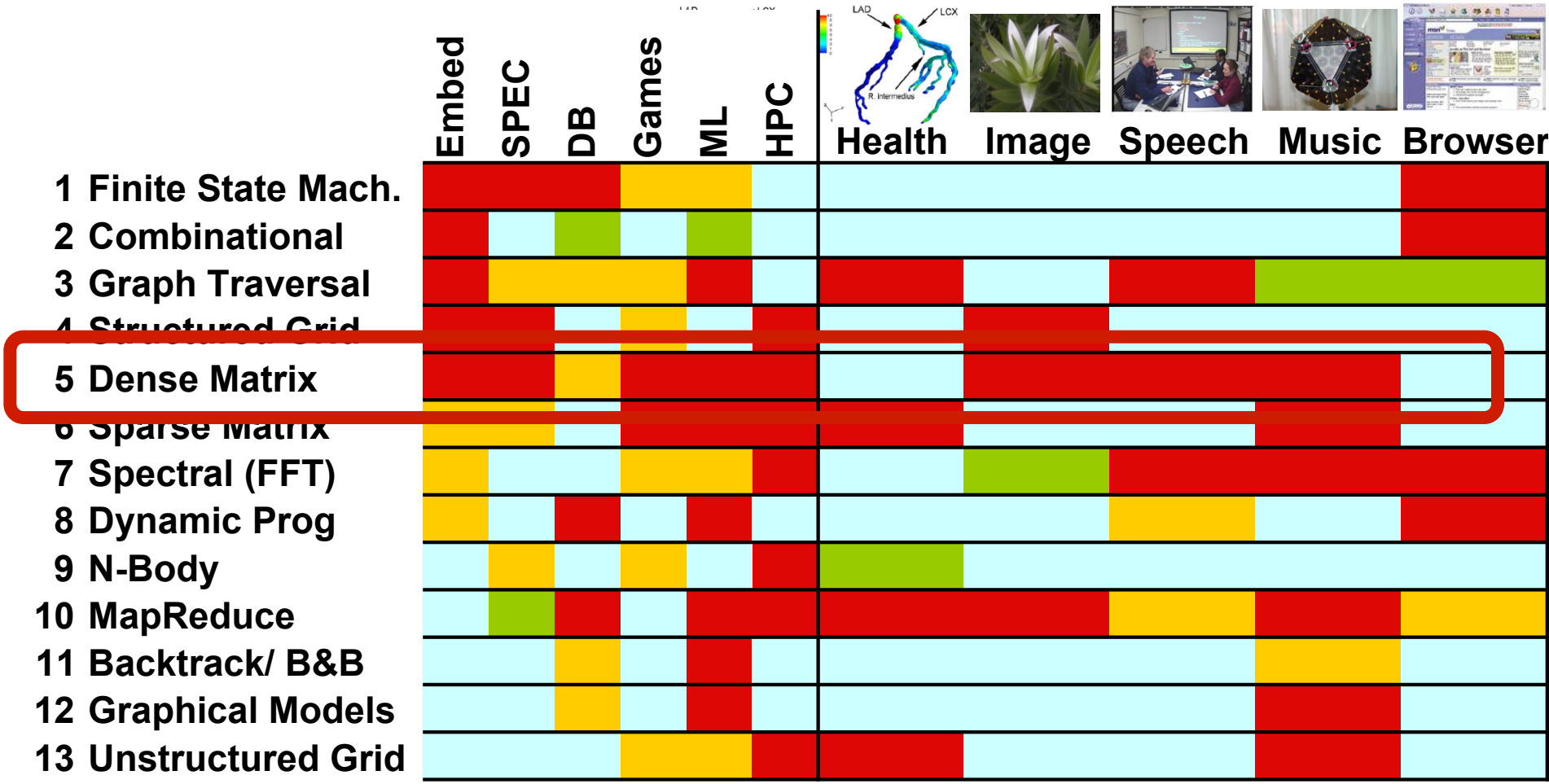
- Idealized and actual costs in modern processors
- Memory hierarchies
- Parallelism within single processors
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Attainable lower bounds on communication
  - Simple cache model
  - Warm-up: Matrix-vector multiplication
  - Naïve vs optimized Matrix-Matrix Multiply
    - Minimizing data movement
    - Beating $O(n^3)$ operations
    - Practical optimizations (*continued next time*)

# Why Matrix Multiplication?

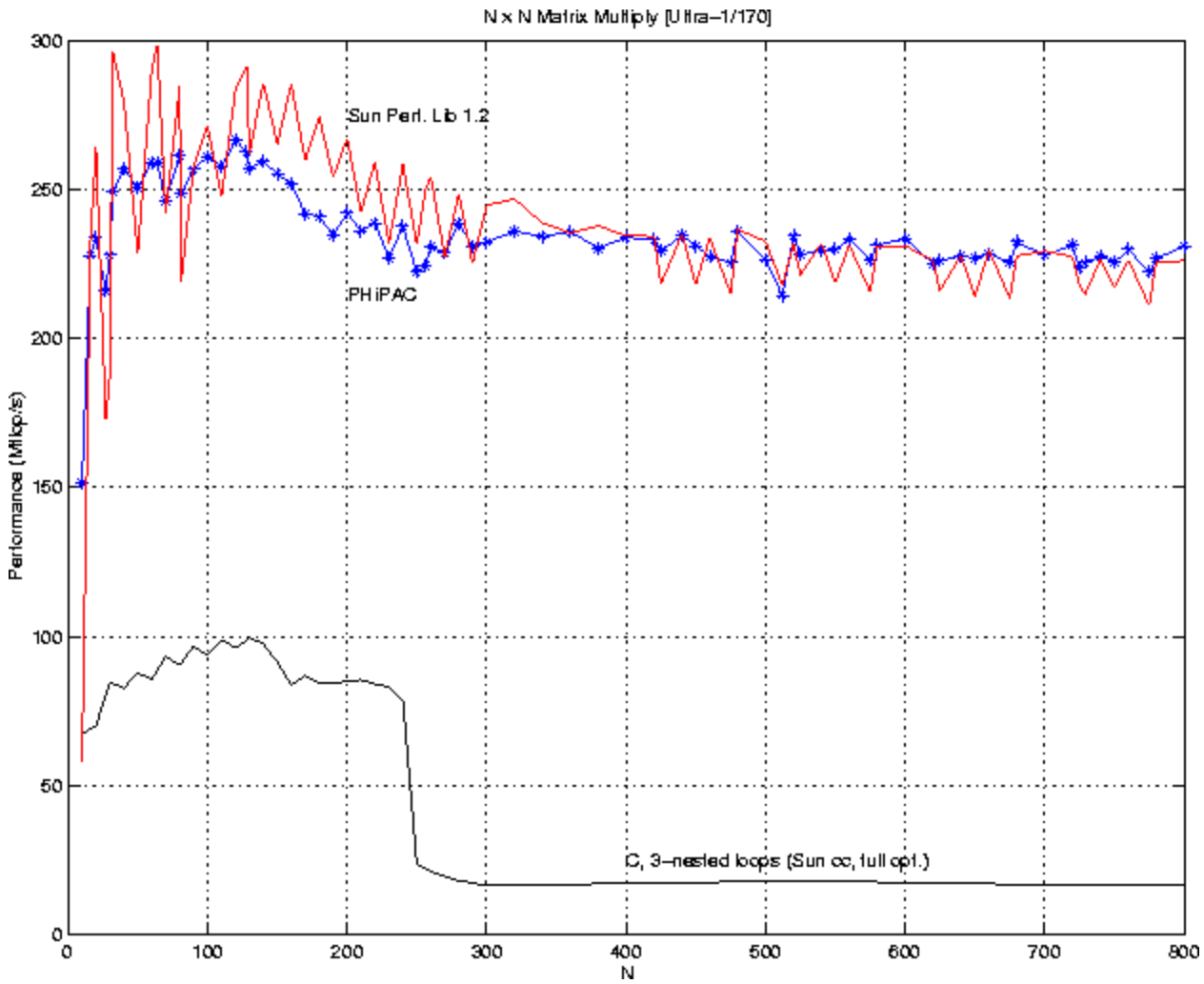
- An important kernel in many problems
  - Appears in many linear algebra algorithms
    - Bottleneck for dense linear algebra, including Top500
  - One of the motifs of parallel computing
  - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
- Optimization 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)



# Matrix-multiply, optimized several ways




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

# 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)

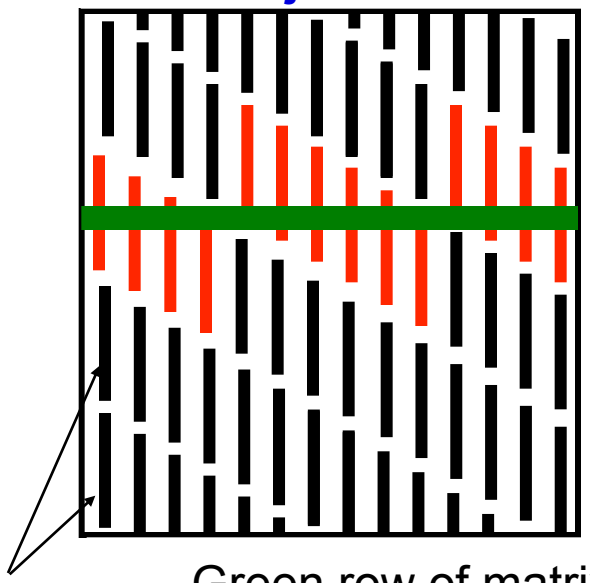
**Column major**

| | | | |
|---|---|---|---|
| 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

Green row of matrix is stored in red cachelines

- Column major (for now)



**Figure source: Larry Carter, UCSD**

# Using a Simple Model of Memory to Optimize

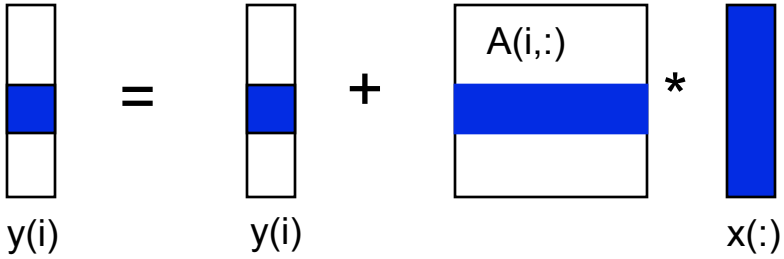
- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - $t_m$ = time per slow memory operation
  - f = number of arithmetic operations
  - $t_f$ = time per arithmetic operation << $t_m$
  - q = f / m  average number of flops per slow memory access
- Minimum possible time = f * $t_f$ when all data in fast memory
- Actual time
  - f * $t_f$ + m * $t_m$ = f * $t_f$ * (1 + $t_m/t_f$ * 1/q)
- Larger q means time closer to minimum f * $t_f$
  - q ≥ $t_m/t_f$ needed to get at least half of peak speed

***Computational Intensity:*** **Key to algorithm efficiency**

***Machine Balance:*** **Key to machine efficiency**

# Warm up: Matrix-vector multiplication

```
{implements y = y + A*x}
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



# Warm up: Matrix-vector multiplication

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
      {read row i of A into fast memory}
      for j = 1:n
            y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- $m$ = number of slow memory refs = $3n + n^2$
- $f$ = number of arithmetic operations = $2n^2$
- $q$ = $f / m \approx 2$

- Matrix-vector multiplication limited by slow memory speed

# Modeling Matrix-Vector Multiplication

- Compute time for nxn = 1000x1000 matrix
- Time

$$f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$$
$$= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$$

- 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 |

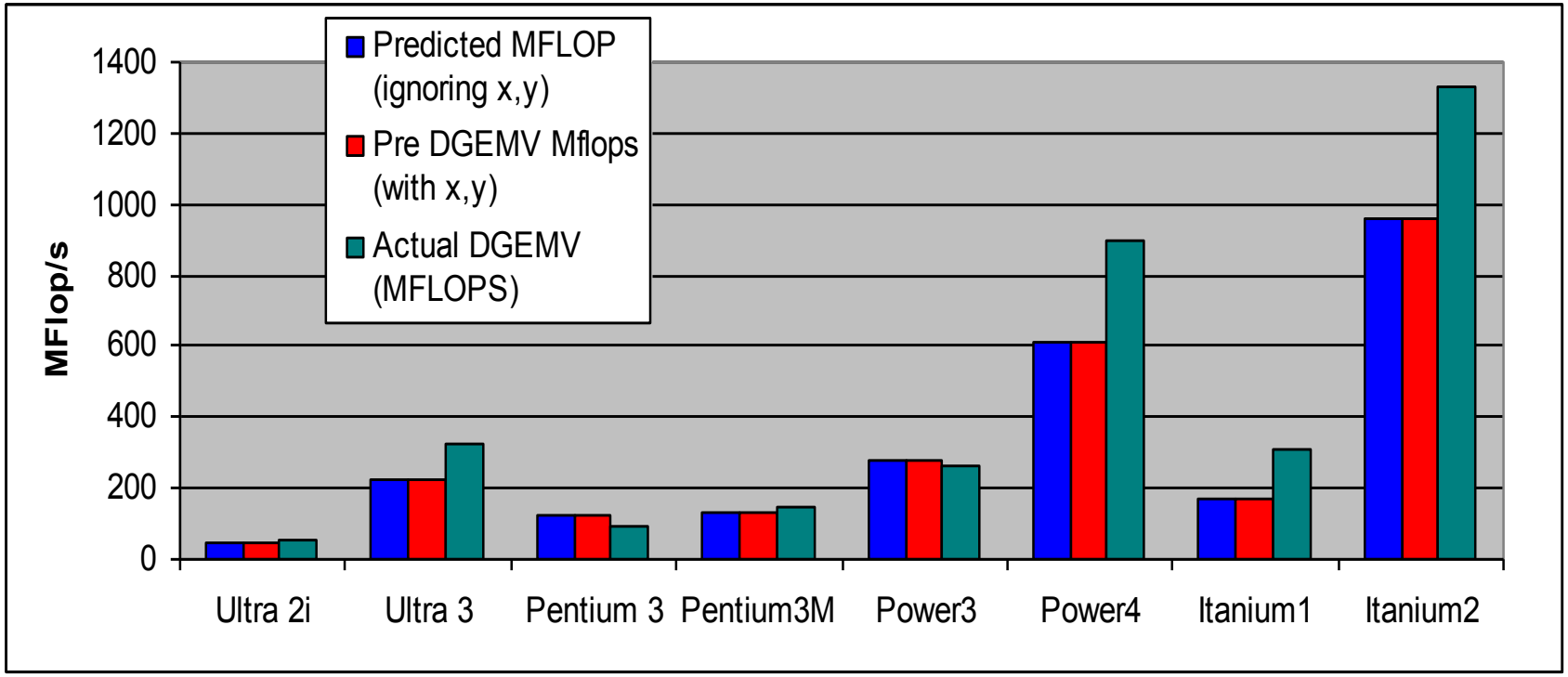
***machine balance* (q must be at least this for ½ peak speed)**

# 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
- Could simplify even further by ignoring memory operations in X and Y vectors
  - Mflop rate/element = $2 / (2^* t_f + t_m)$

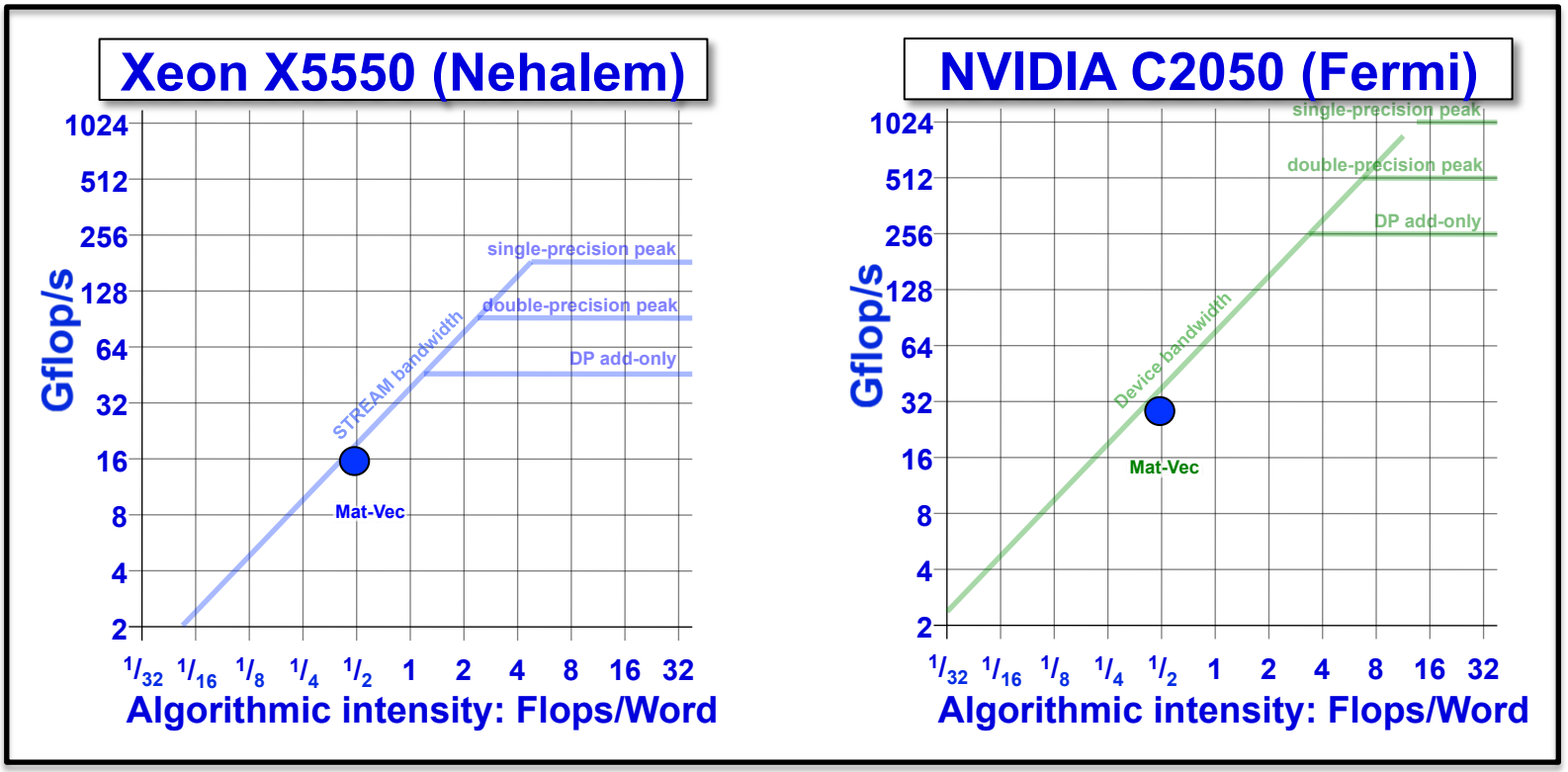
# Validating the Model

- 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 on most recent ones due to latency estimate



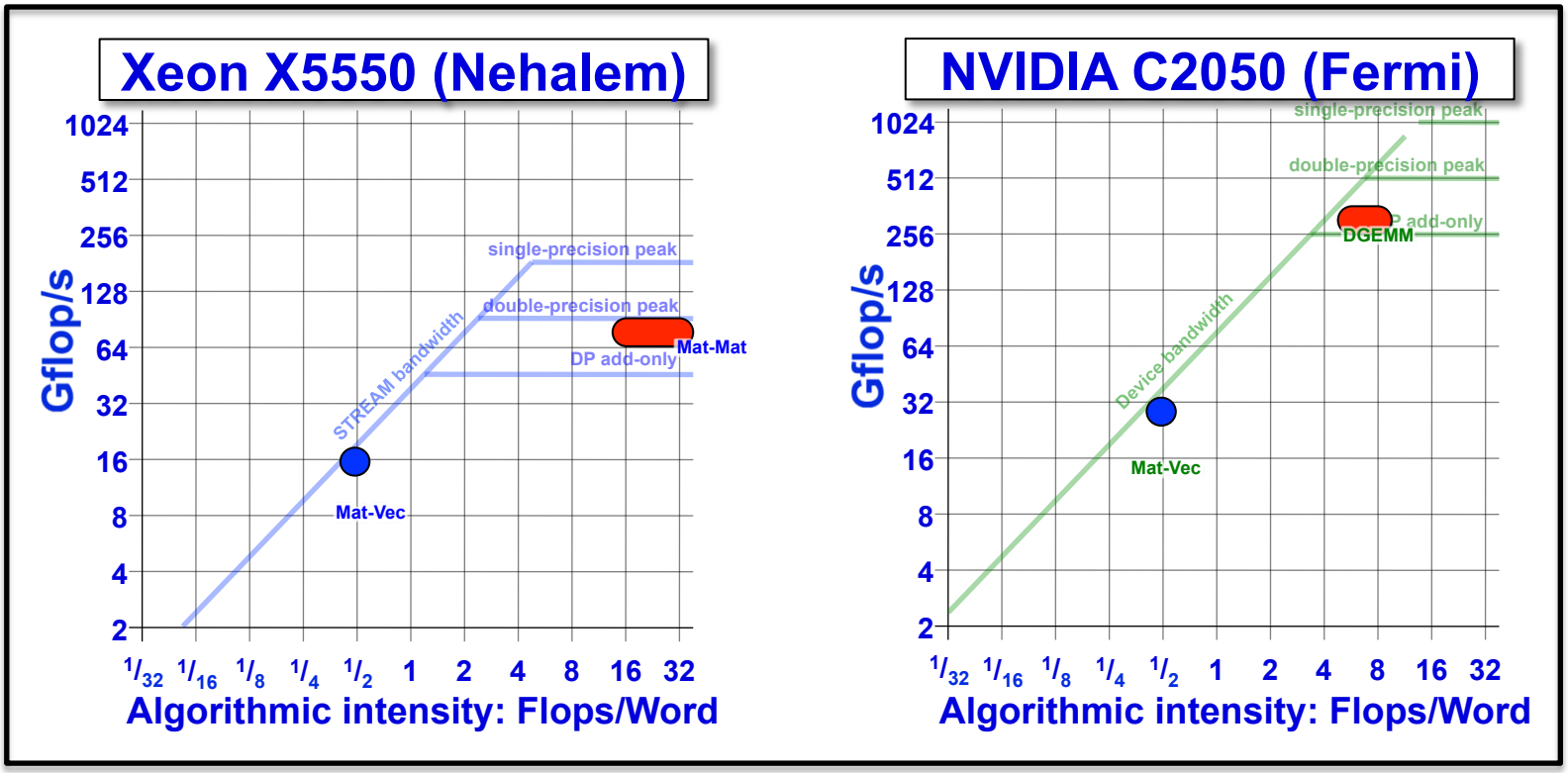
# Roofline for Matrix-Vector multiply

- Roofline capture the gross differences between BLAS2 (matrix-vector) and BLAS3 (Matrix-Matrix)



# Roofline for Matrix-Vector and Matrix-Matrix multiply

- Roofline capture the gross differences between BLAS2 (matrix-vector) and BLAS3 (Matrix-Matrix)

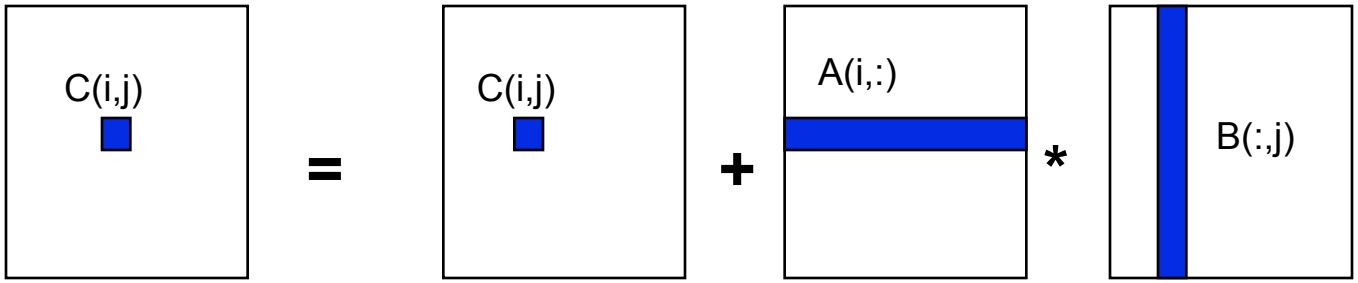


# Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
      for j = 1 to n
            for k = 1 to n
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

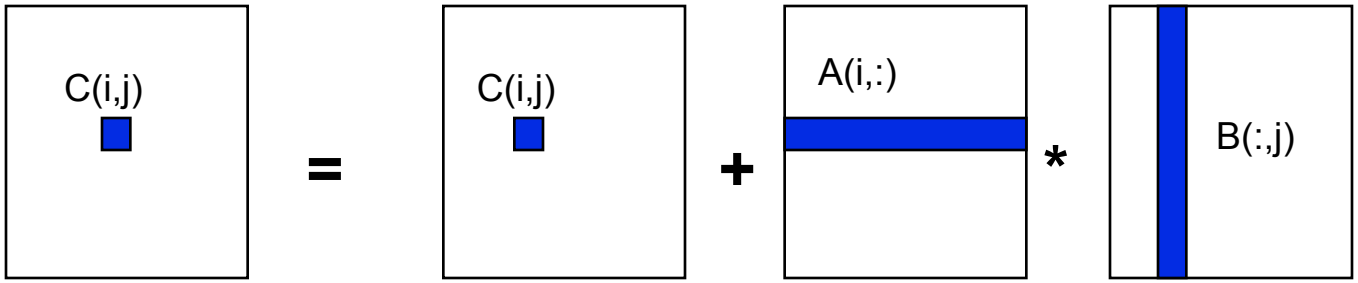
Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory

q potentially as large as $2*n^3 / 3*n^2 = O(n)$



# Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



# Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$ to read each column of B $n$ times

$+ n^2$ to read each row of A once

$+ 2n^2$ to read and write each element of C once

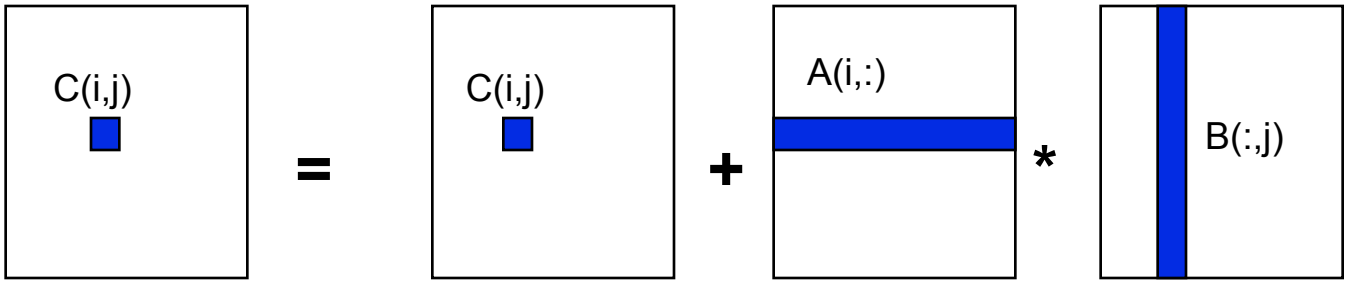
$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$

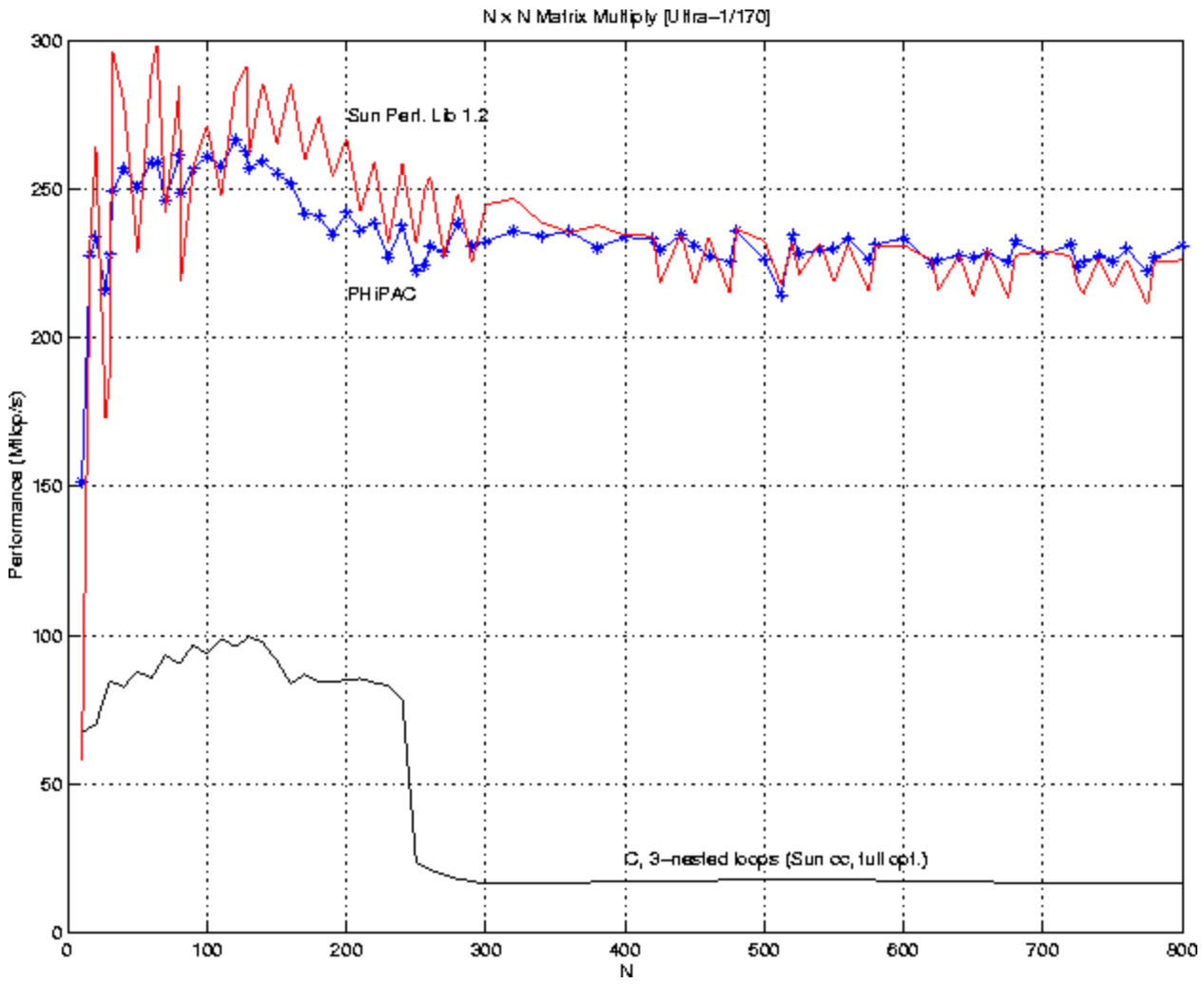
$\approx 2$ for large $n$, no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row i of A times B

Similar for any other order of 3 loops



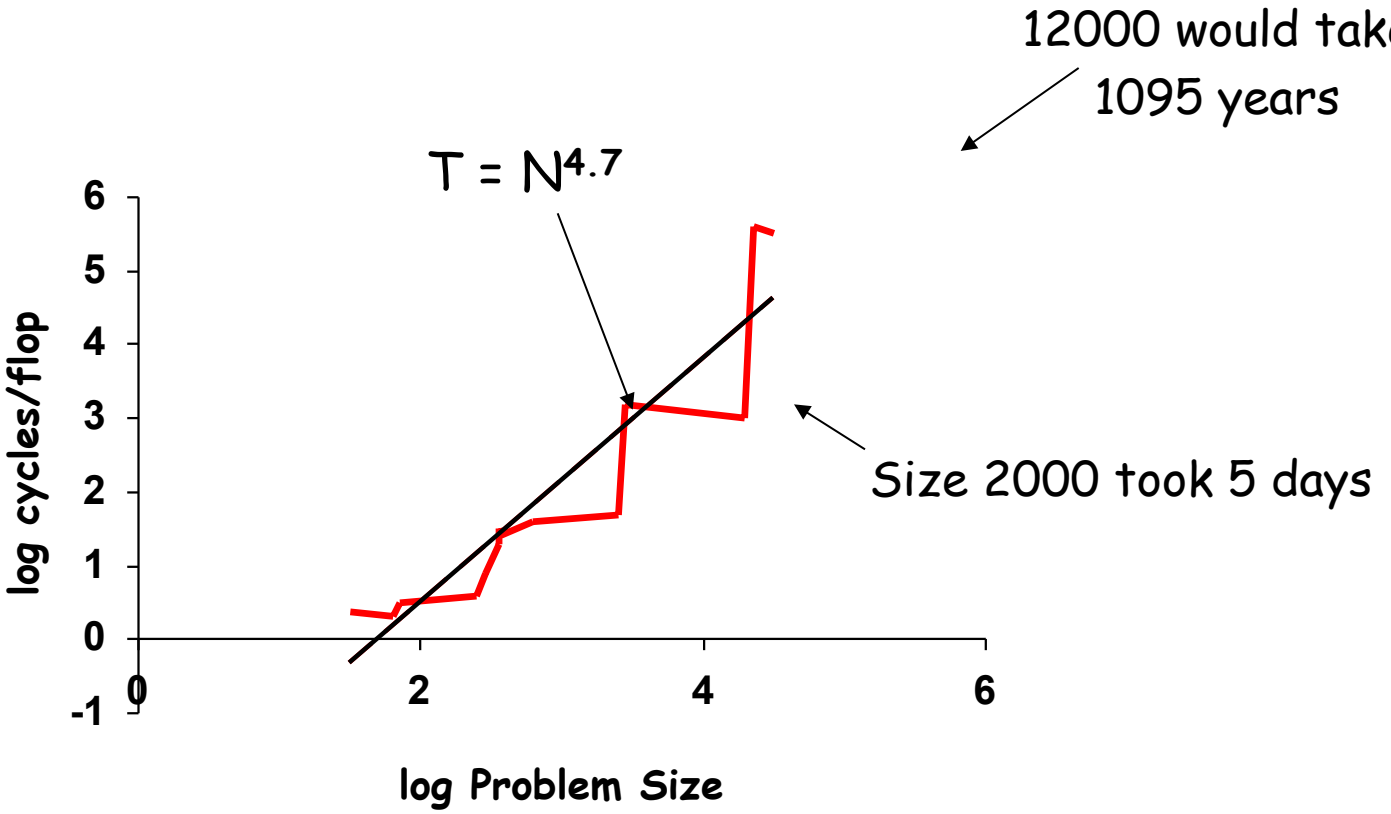
# Matrix-multiply, optimized several ways




**CS267-inspired PHiPAC project**

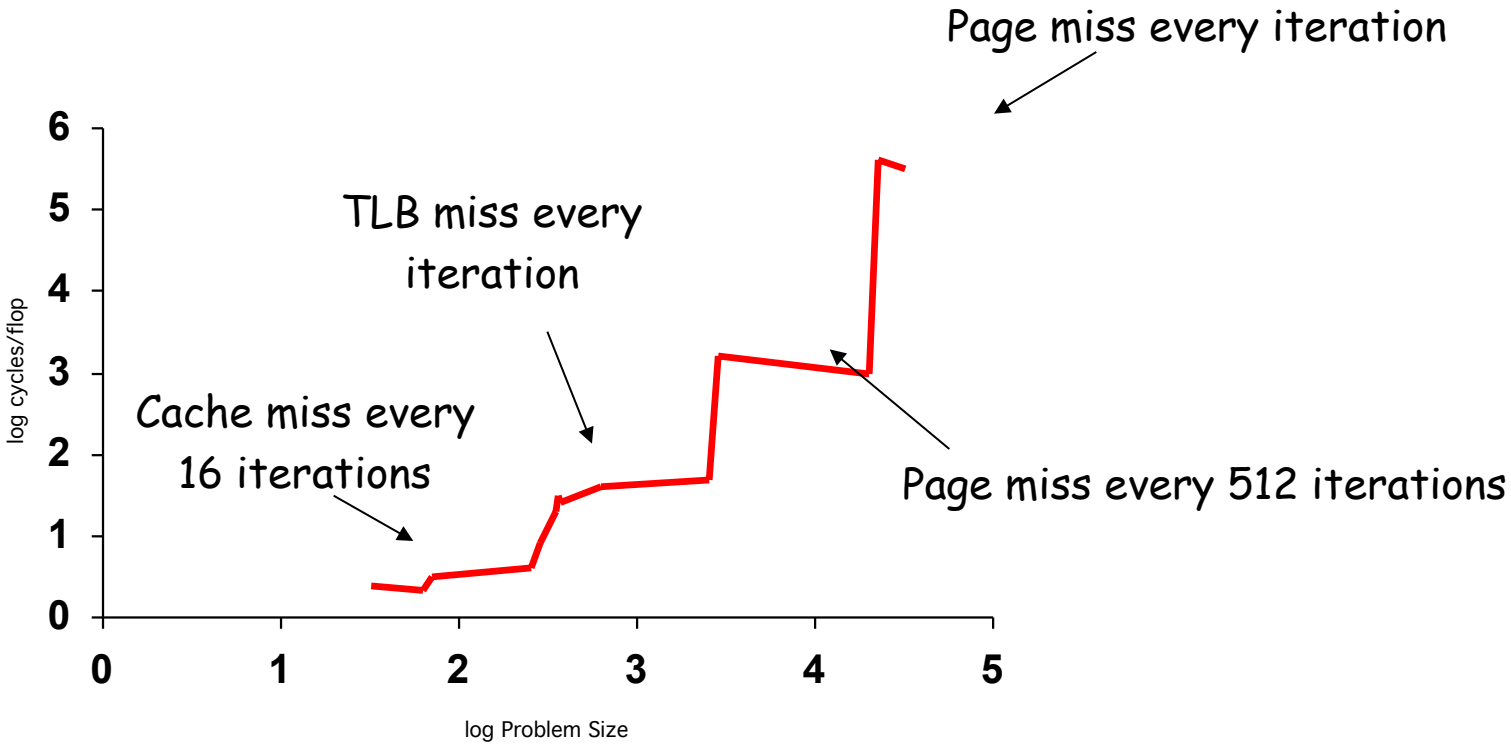
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

# 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



# Blocked (Tiled) Matrix Multiply

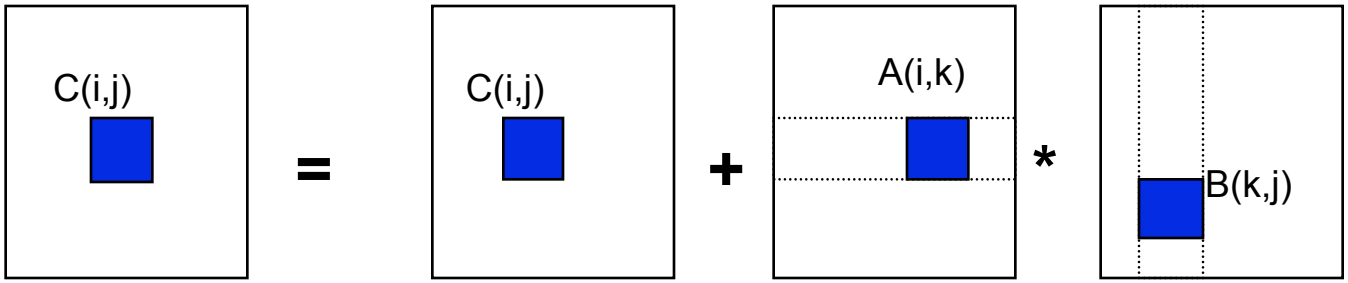
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
b=n / N is called the block size

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```

**cache does this automatically**

**3 nested loops inside**

**block size = loop bounds**




Tiling for registers (managed by you/compiler) or caches (hardware)

# Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between slow and fast memory

matrix has nxn elements, and NxN blocks each of size bxb

f is number of floating point operations, $2n^3$ for this problem

q = f / m is our measure of algorithm efficiency in the memory system

So:

m = $N*n^2$ read each block of B $N^3$ times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

+ $N*n^2$ read each block of A $N^3$ times

+ $2n^2$ read and write each block of C once

= $(2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\approx n / N = b$ for large n

So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply (q=2)

# Using Analysis to Understand Machines

The blocked algorithm has computational intensity $q \approx b$

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size $M_{fast}$

$$3b^2 \le M_{fast}, \quad \text{so} \quad q \approx b \le (M_{fast}/3)^{1/2}$$

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size

$$M_{fast} \ge 3b^2 \approx 3q^2 = 3(t_m/t_f)^2$$

- This size is reasonable for L1 cache, but not for register sets
- Note: analysis assumes it is possible to schedule the instructions perfectly

| | | **required** |
|---|---|---|
| | **t_m/t_f** | **KB** |
| Ultra 2i | 24.8 | 14.8 |
| Ultra 3 | 14 | 4.7 |
| Pentium 3 | 6.25 | 0.9 |
| Pentium3M | 10 | 2.4 |
| Power3 | 8.75 | 1.8 |
| Power4 | 15 | 5.4 |
| Itanium1 | 36 | 31.1 |
| Itanium2 | 5.5 | 0.7 |

# Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each C[i,j] by applying commutativity and associativity
  - Get slightly different answers from naïve code, because of roundoff - OK
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \approx b \leq (M_{fast}/3)^{1/2}$$

- There is a lower bound result that says we cannot do any better than this (using only associativity, so still doing $n^3$ multiplications)

- **Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O( (M_{fast})^{1/2} )$**
  - **#words moved between fast and slow memory $= \Omega (n^3 / (M_{fast})^{1/2} )$**

# Communication lower bounds for Matmul

- Hong/Kung theorem is a lower bound on amount of data communicated by matmul
  - Number of words moved between fast and slow memory (cache and DRAM, or DRAM and disk, or …) = $\Omega\ (n^3 / M_{fast}^{1/2})$
- Cost of moving data may also depend on the number of "messages" into which data is packed
  - Eg: number of cache lines, disk accesses, …
  - #messages = $\Omega\ (n^3 / M_{fast}^{3/2})$
- Lower bounds extend to anything "similar enough" to 3 nested loops
  - Rest of linear algebra (solving linear systems, least squares…)
  - Dense and sparse matrices
  - Sequential and parallel algorithms, …
- More recent: extends to any nested loops accessing arrays
- Need (more) new algorithms to attain these lower bounds…

# Summary of Lecture 2

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - Before you parallelize, make sure you're getting good serial performance
  - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
  - Pipelining, SIMD, etc
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby to recently used data
- Can rearrange code/data to improve locality
  - Goal: minimize communication = data movement

# Class Logistics

- Homework 0 posted on web site
  - Find and describe interesting application of parallelism
  - Due Wednesday Jan 24
  - Could even be your intended class project
  - We will add undergrads to bcourses so you can submit
- Fill in on-line class survey by midnight
  - We need this to assign teams for Homework 1
  - Teams will be announced no later than Monday
  - HW 1 will be posted by then