

---

**CS267**  
**Lecture 2**  
**Single Processor Machines:**  
**Memory Hierarchies**  
**and Processor Features**

**Case Study: Tuning Matrix Multiply**

James Demmel

[http://www.cs.berkeley.edu/~demmel/cs267\\_Spr12/](http://www.cs.berkeley.edu/~demmel/cs267_Spr12/)

# 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

# **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 use fast software that someone else has written and hidden all these details from me so I don’t have to worry about them!”
- All of the above, at different points in time

# 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
  - Attainable lower bounds on communication

# 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
  - Attainable lower bounds on communication

# 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

$A = B + C \Rightarrow$

- 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

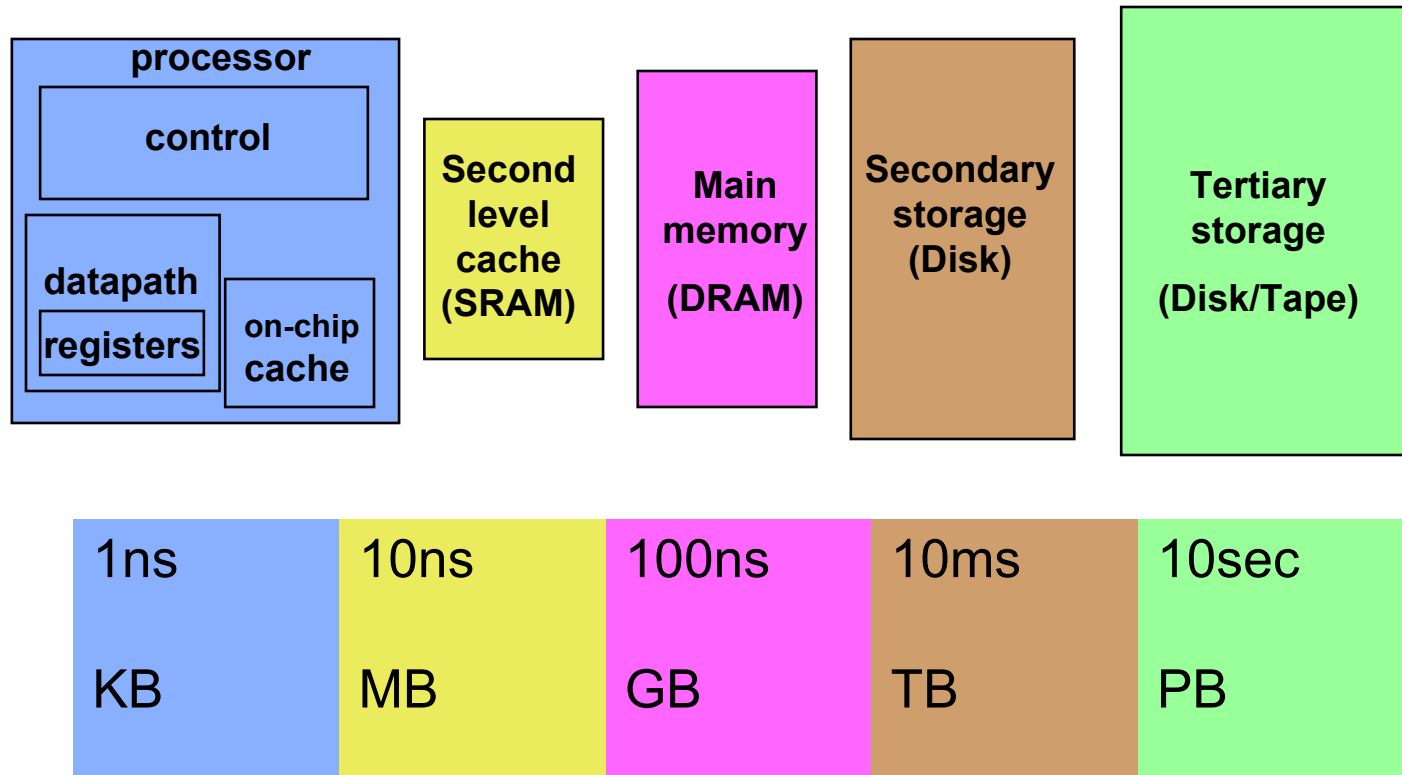
---

- 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
  - Use of performance models to understand performance
  - Attainable lower bounds on communication



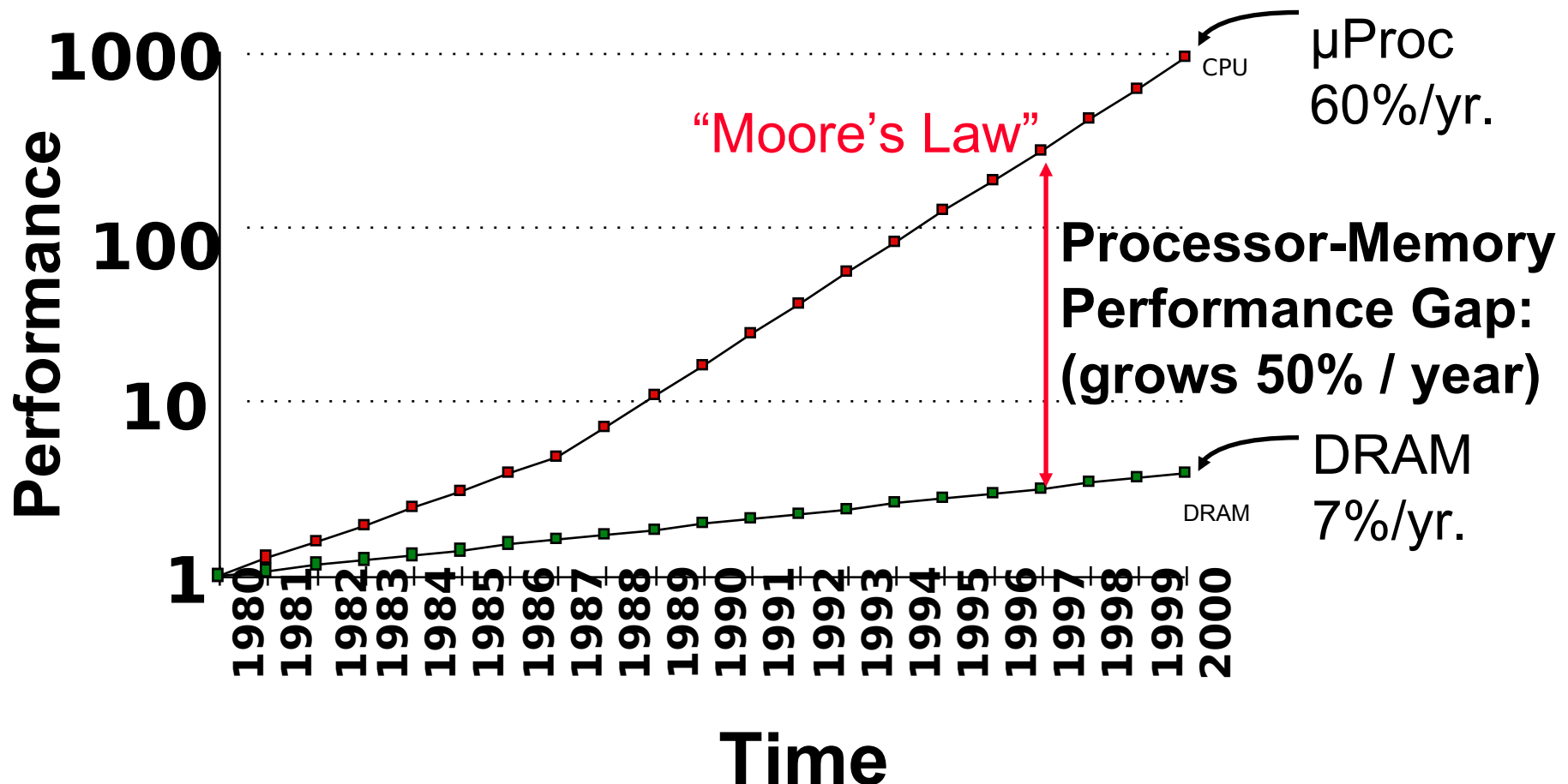
# Memory Hierarchy

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



# Processor-DRAM Gap (latency)

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory



# 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**

# Cache Basics

---

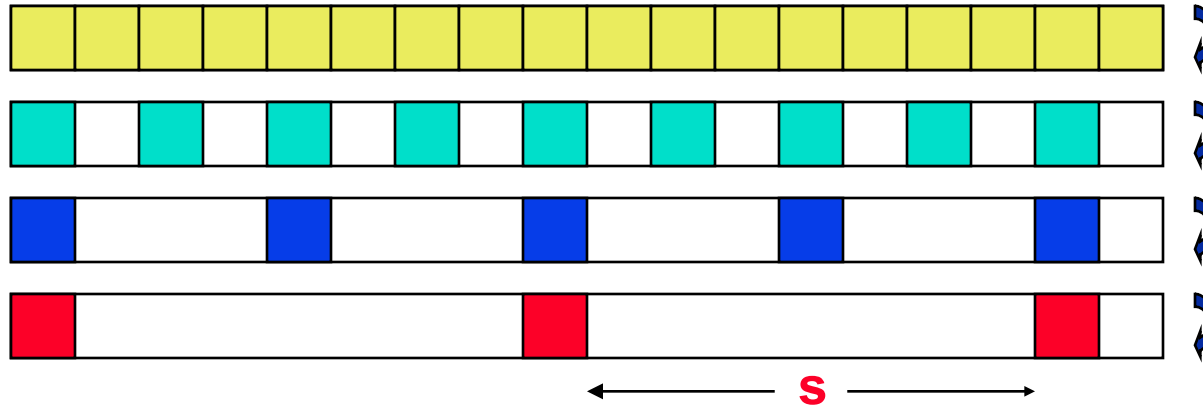
- **Cache** is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
  - Need 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**
  - **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), ...**
  - **And it isn't always a hierarchy**

# Experimental Study of Memory (Membench)

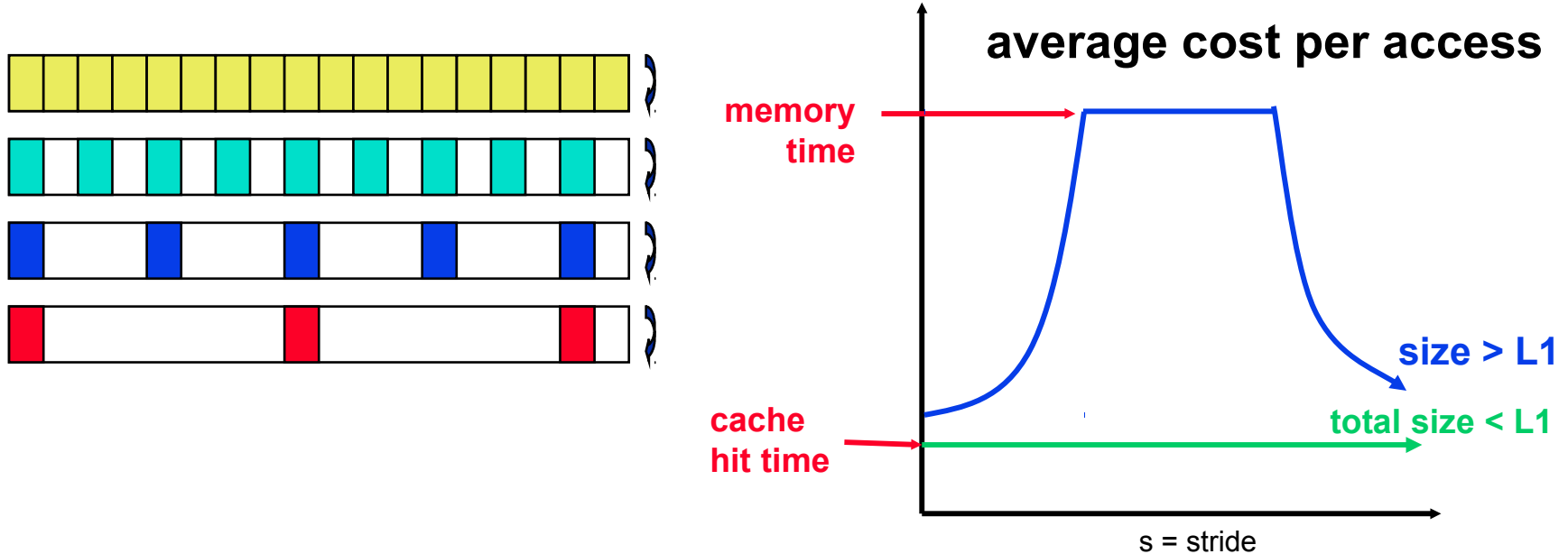
- Microbenchmark for memory system performance



- 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)

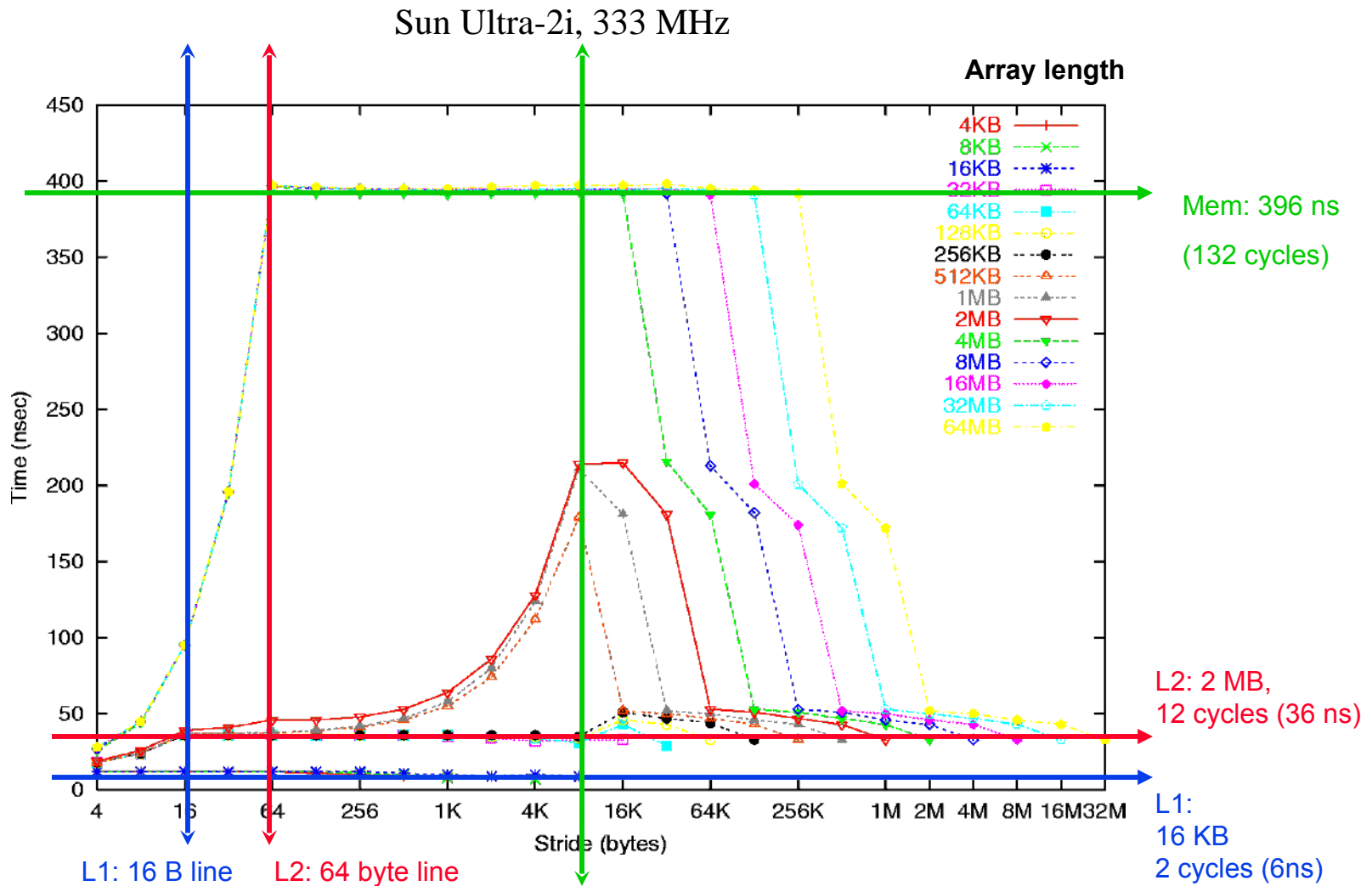
1 experiment

# Membench: What to Expect



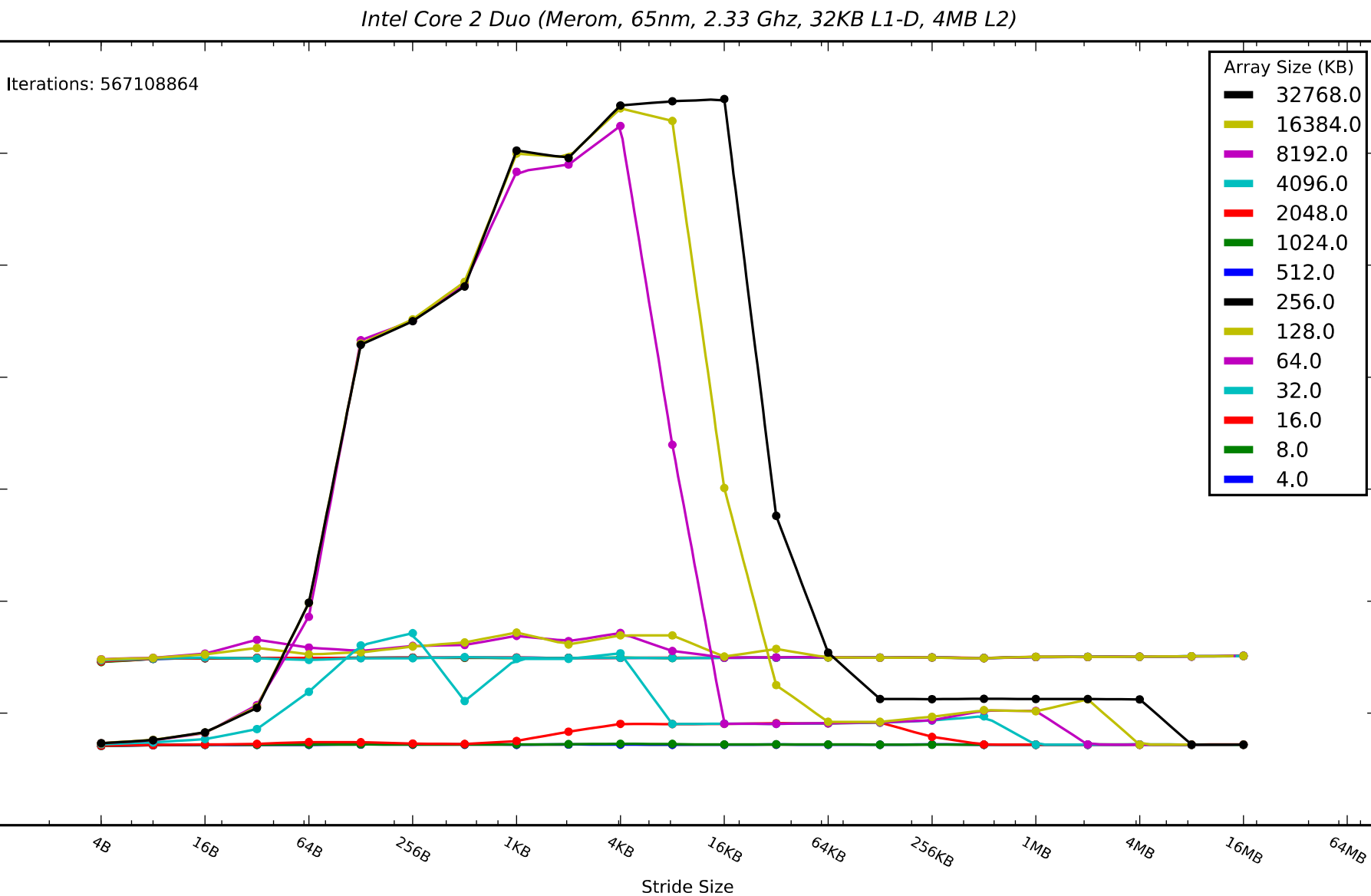
- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most  $\frac{1}{4}$  miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large 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](http://www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps) for details





# Memory Hierarchy on a Power3 (Seaborg)

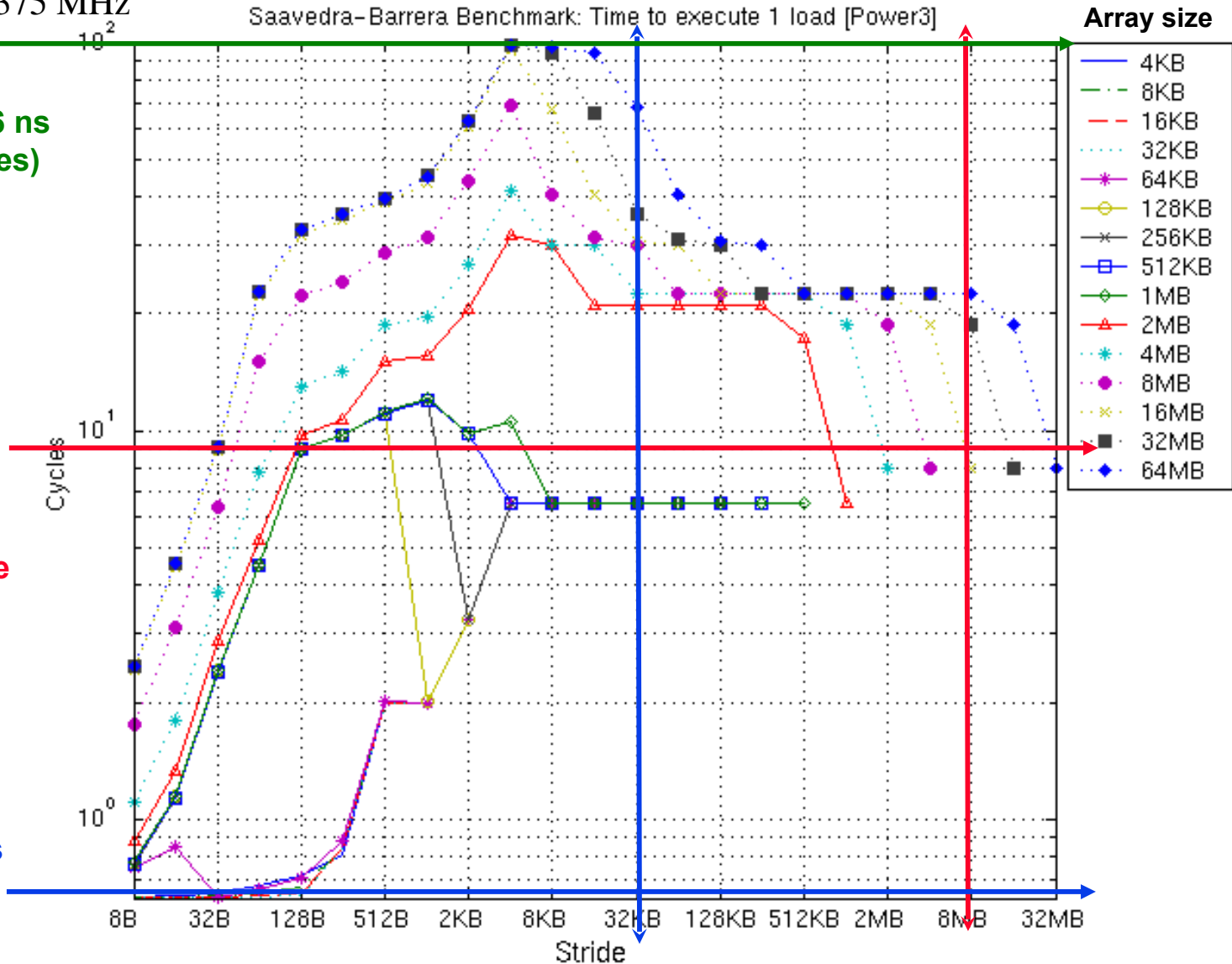
Power3, 375 MHz

Saavedra-Barrera Benchmark: Time to execute 1 load [Power3]

Mem: 396 ns  
(132 cycles)

L2: 8 MB  
128 B line  
9 cycles

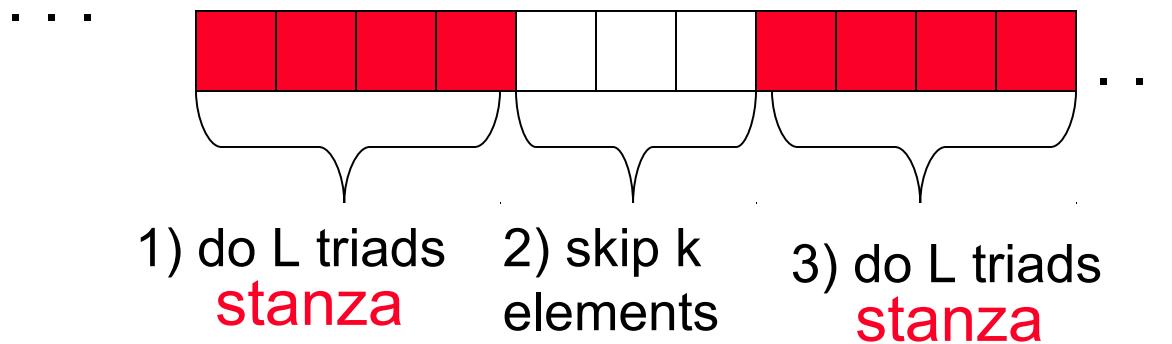
L1: 32 KB  
128B line  
.5-2 cycles



# Stanza Triad

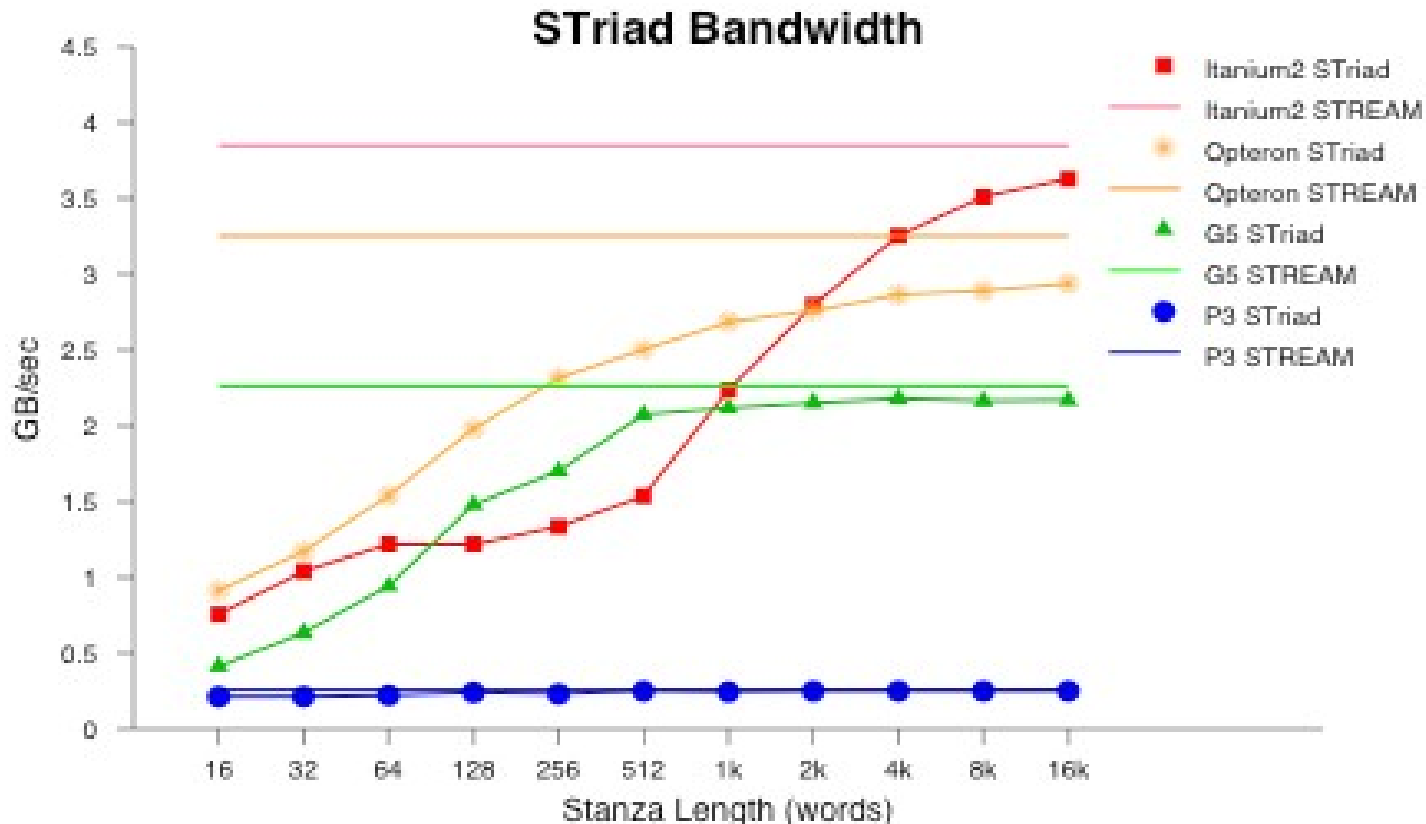
- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- **Stanza (L)** is the length of a unit stride run

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



# Stanza Triad

F



- This graph (x-axis) starts at a cache line size ( $\geq 16$  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](http://crd-legacy.lbl.gov/~oliker/papers/msp_2005.pdf)

# 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

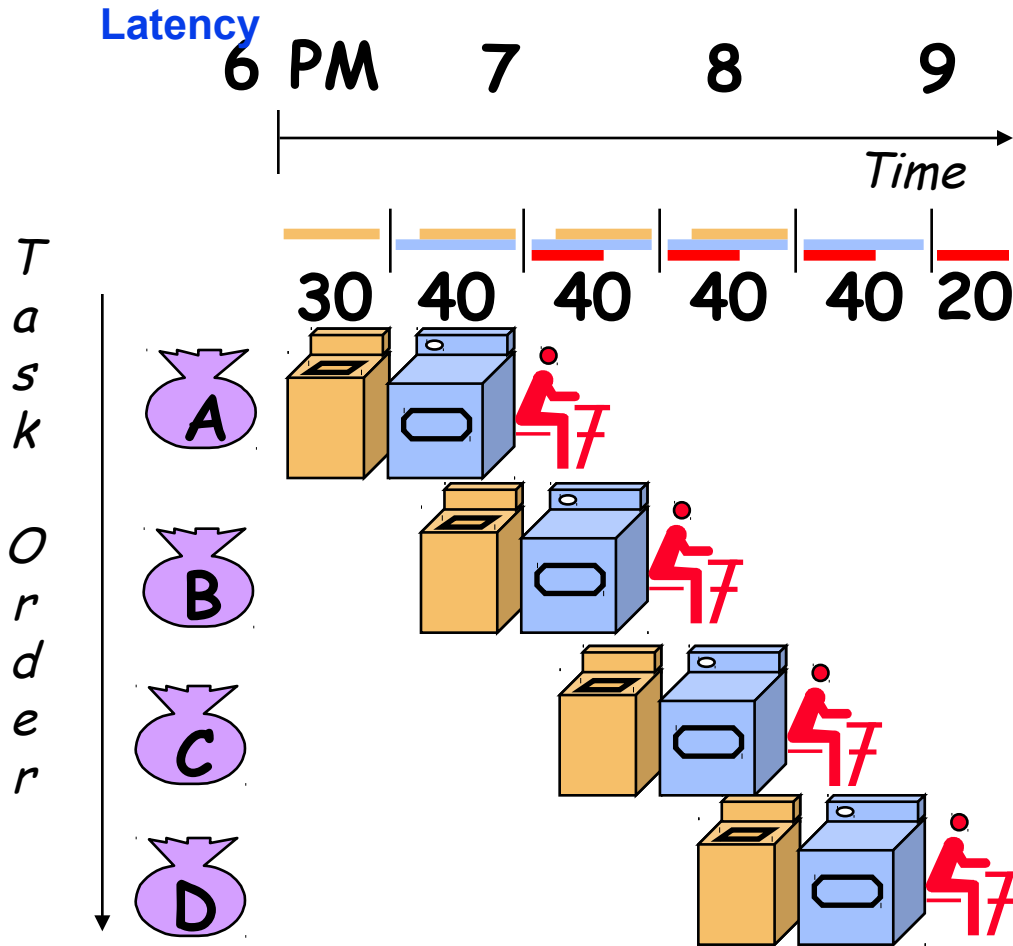
---

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
  - Hidden from software (sort of)
  - Pipelining
  - SIMD units
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Attainable lower bounds on communication

# What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry

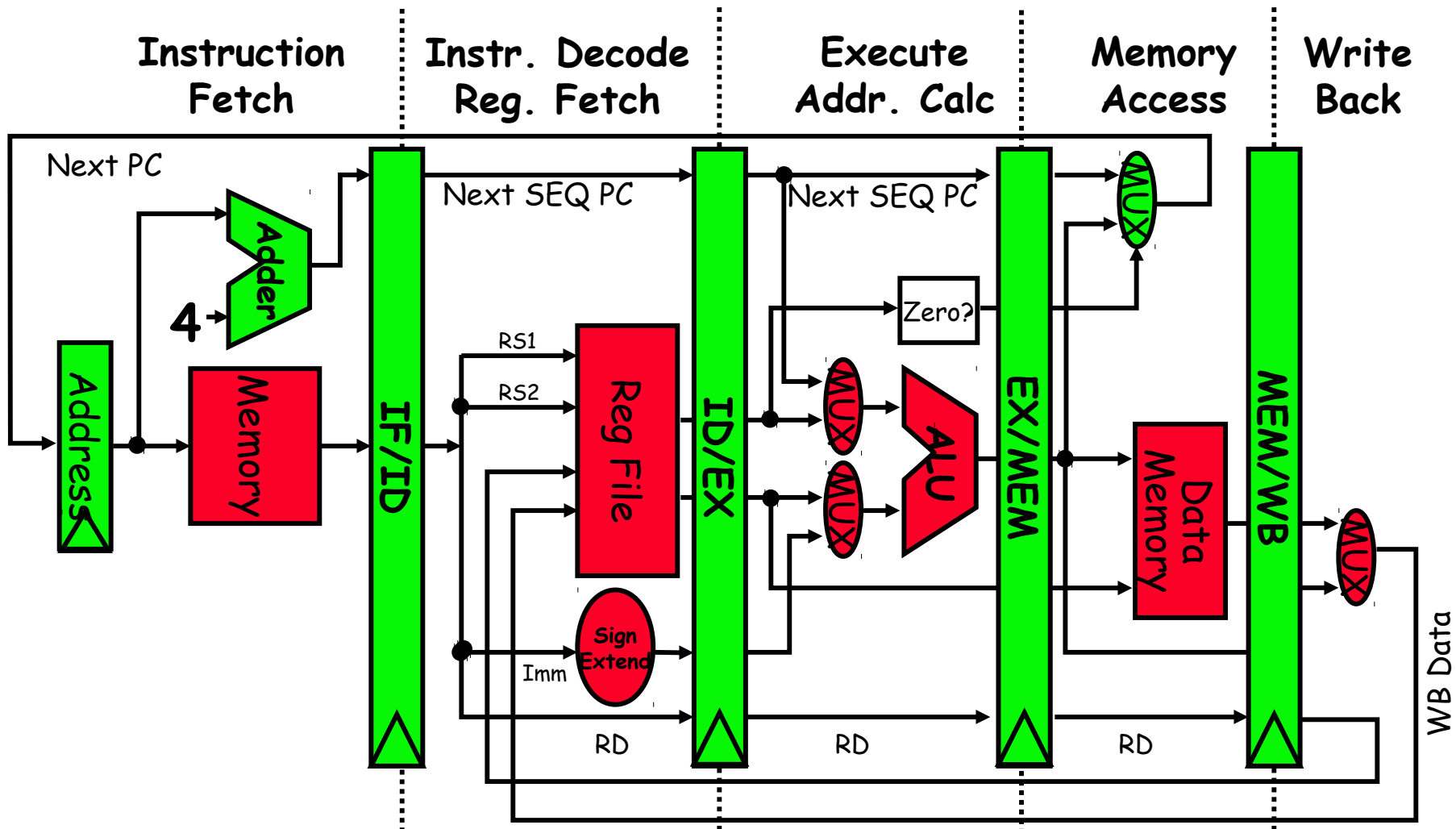
wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- In this example:
  - Sequential execution takes  $4 * 90\text{min} = 6 \text{ hours}$
  - Pipelined execution takes  $30 + 4 * 40 + 20 = 3.5 \text{ hours}$
- Bandwidth = loads/hour
- BW =  $4/6$  l/h w/o pipelining
- BW =  $4/3.5$  l/h w pipelining
- BW  $\leq 1.5$  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



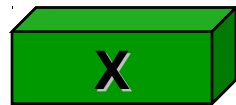
- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle



# SIMD: Single Instruction, Multiple Data

---

- Scalar processing
  - traditional mode
  - one operation produces one result



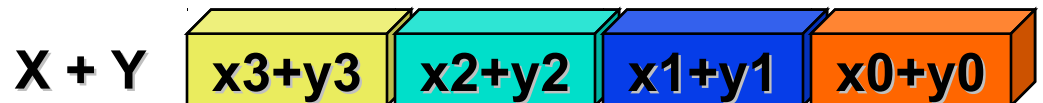
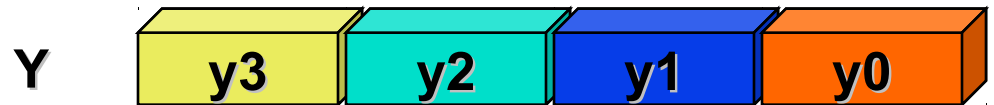
+



- SIMD processing
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results



+

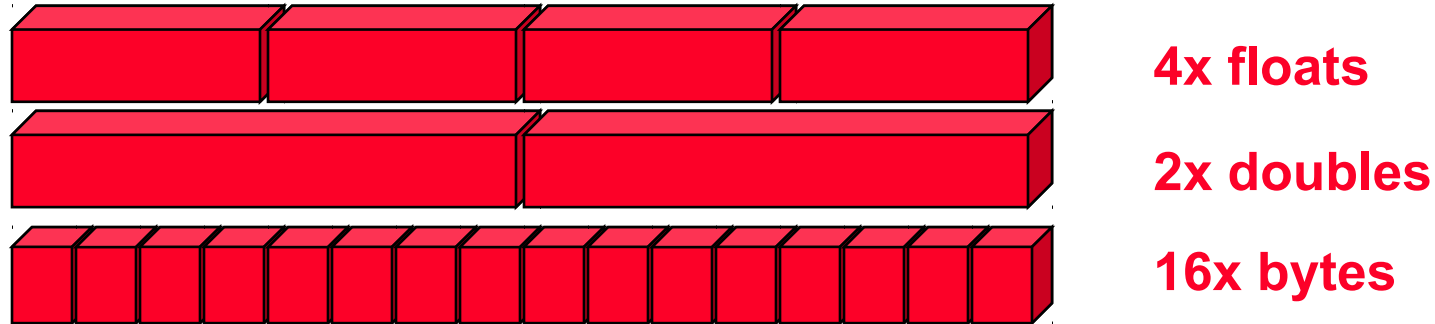


Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

# SSE / SSE2 SIMD on Intel

---

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



- 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
  - 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*)

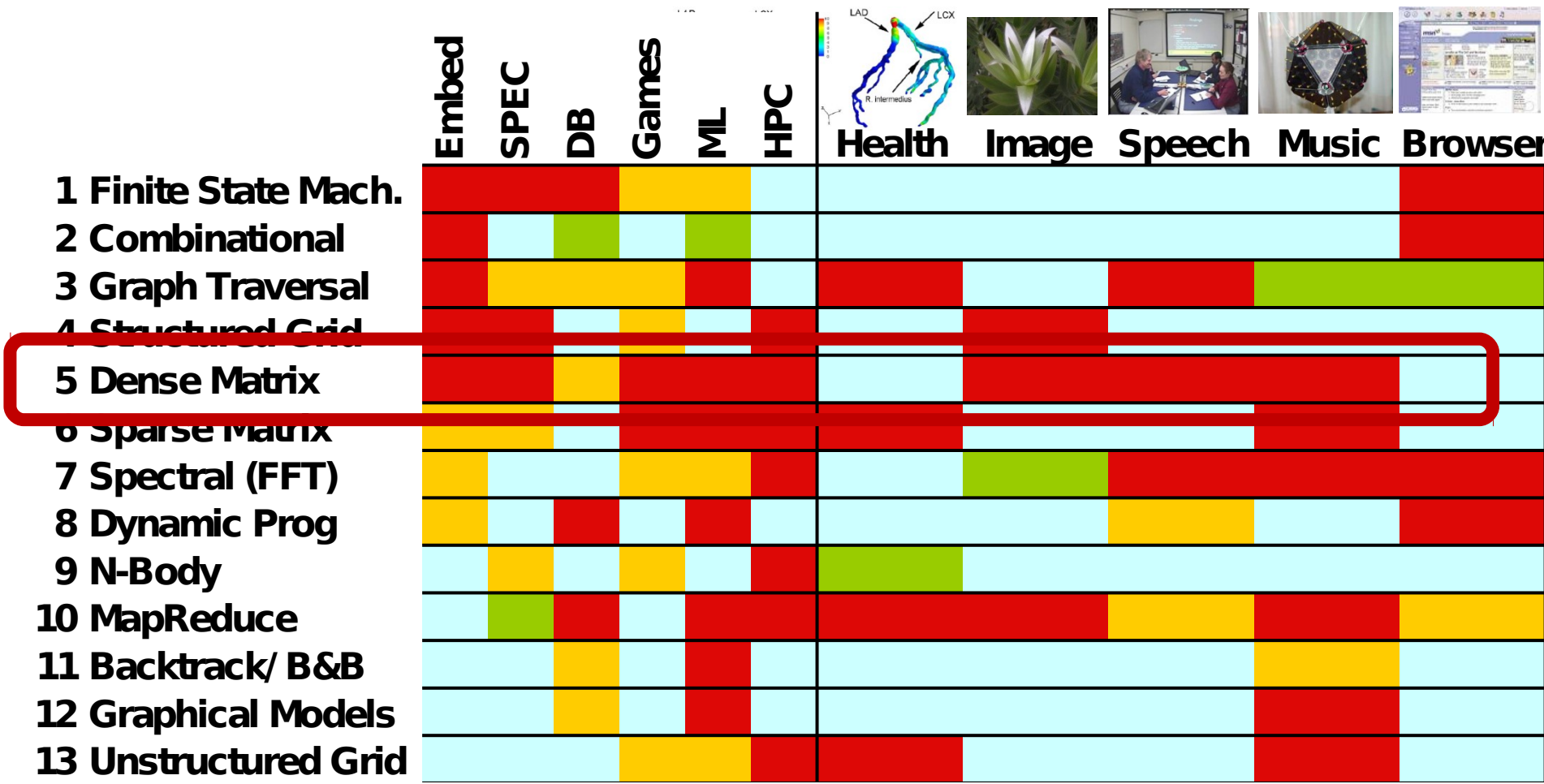
# Why Matrix Multiplication?

---

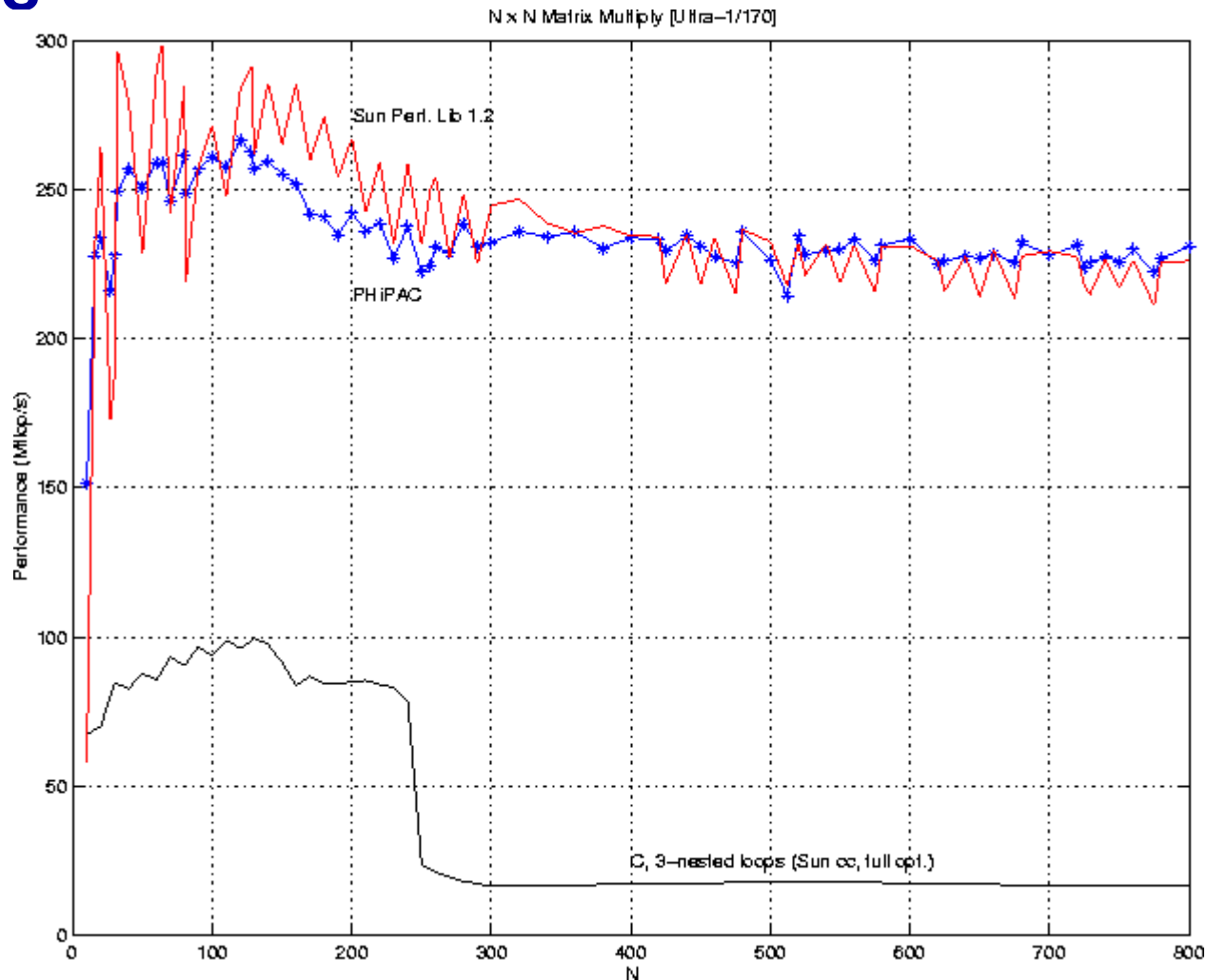
- An important kernel in many problems
  - Appears in many linear algebra algorithms
    - Bottleneck for dense linear algebra
  - One of the 7 dwarfs / 13 motifs of parallel computing
  - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
- Optimization ideas 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**

↓

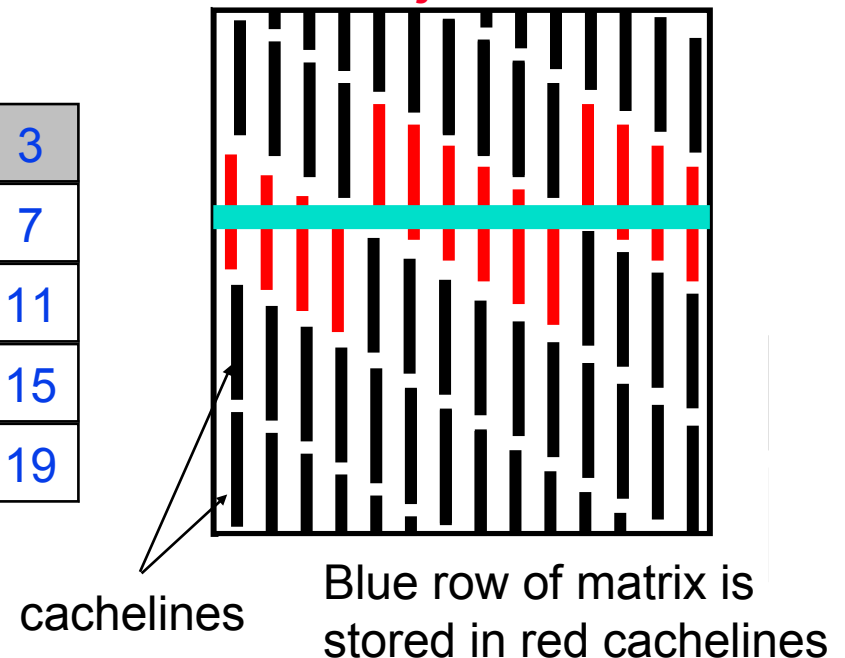
0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

**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



- Column major (for now)



# 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  $\ll t_m$
  - $q = f / m$  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  $f * t_f$ 
  - $q \geq t_m/t_f$  needed to get at least half of peak speed

**Computational Intensity:** Key to algorithm efficiency

**Machine Balance:** Key to machine efficiency

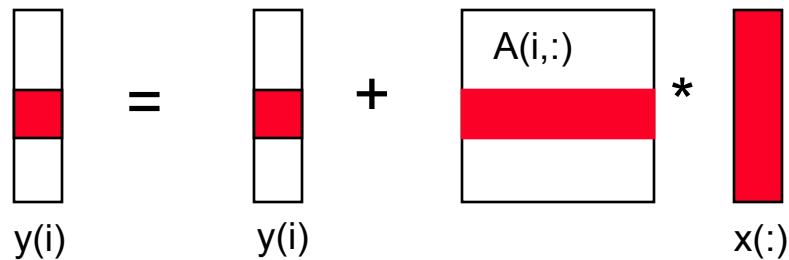
# Warm up: Matrix-vector multiplication

```
{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  $n \times n = 1000 \times 1000$  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 <sub>m</sub> /t <sub>f</sub>
	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)*

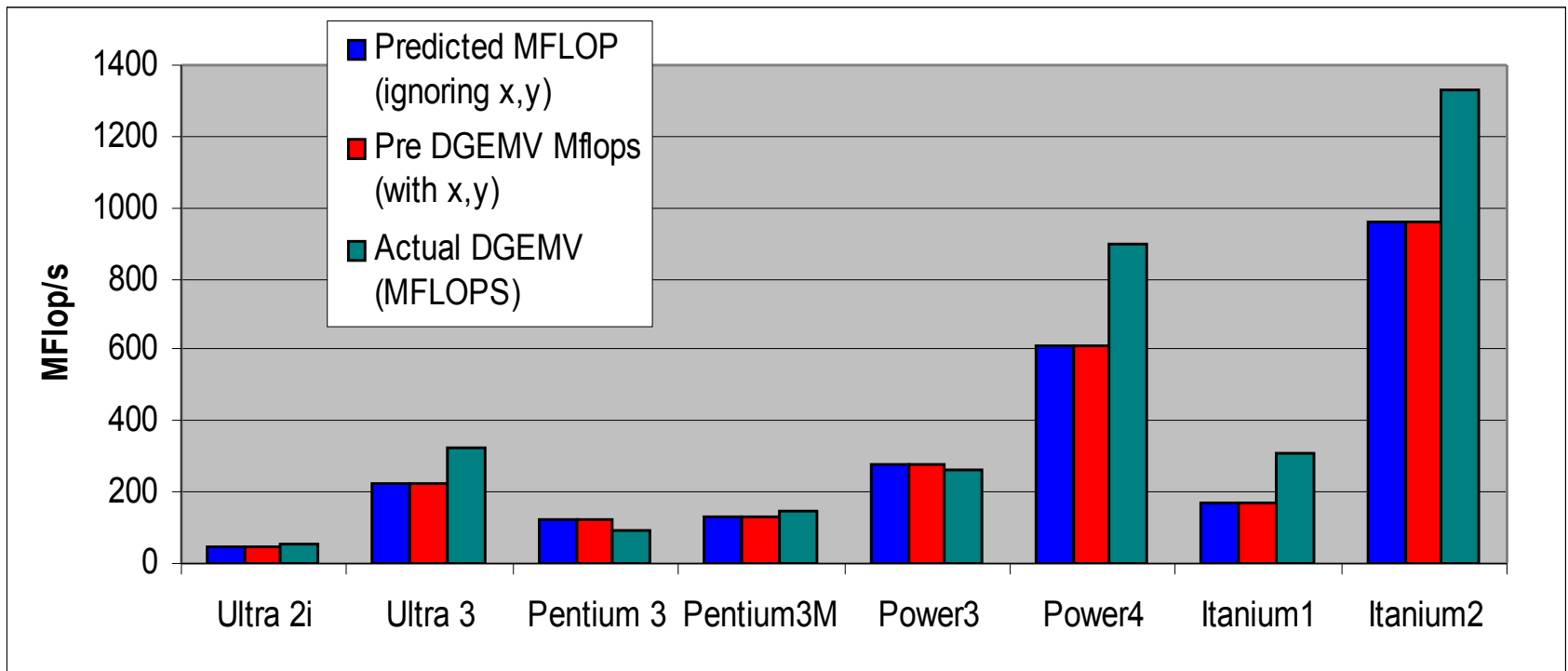
# Simplifying Assumptions

---

- 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



# Naïve Matrix Multiply

```
{implements C = C + A*B}
```

```
for i = 1 to n
```

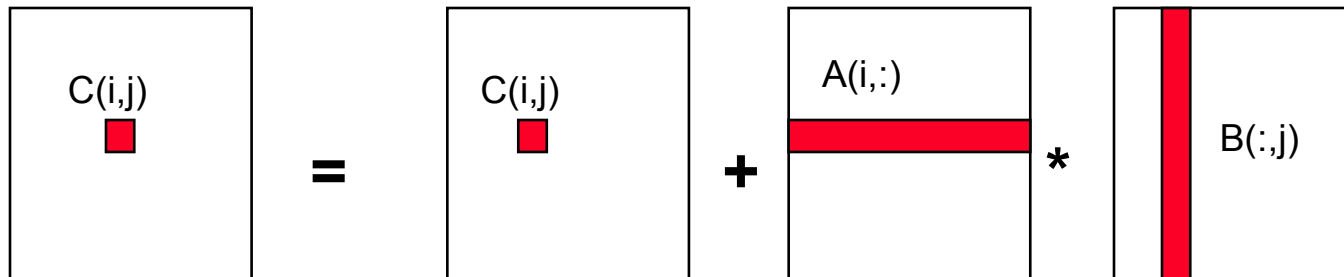
```
  for j = 1 to n
```

```
    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  $n$

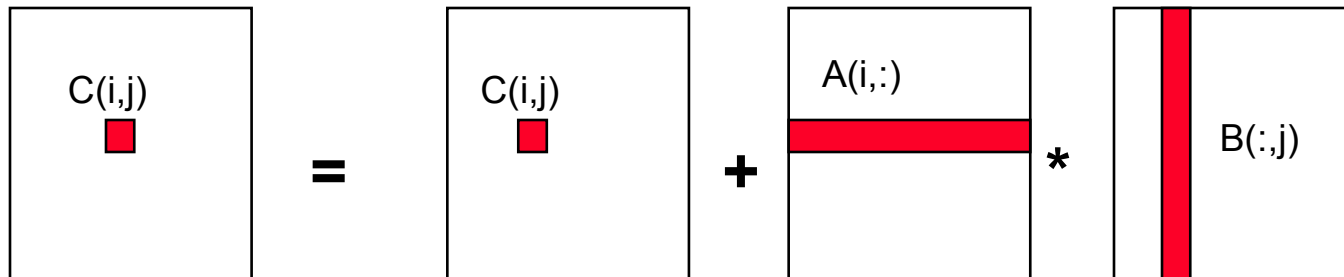
{read  $C(i,j)$  into fast memory}

{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

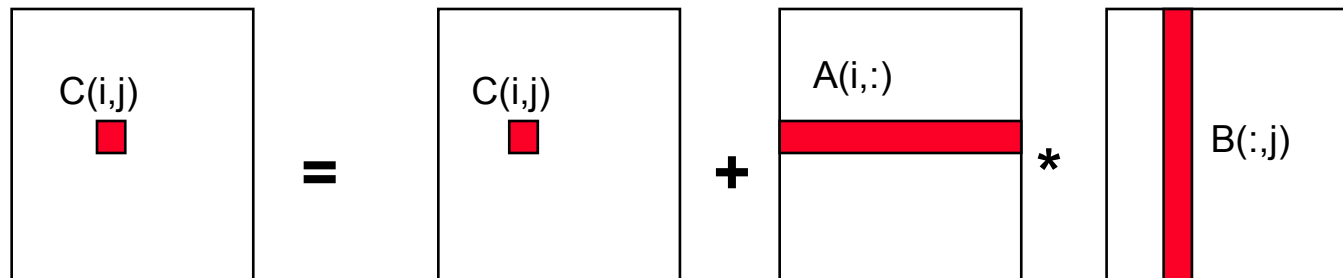
Number of slow memory references on unblocked matrix multiply

$$\begin{aligned} m &= n^3 && \text{to read each column of B } n \text{ times} \\ &+ n^2 && \text{to read each row of A once} \\ &+ 2n^2 && \text{to read and write each element of C once} \\ &= n^3 + 3n^2 \end{aligned}$$

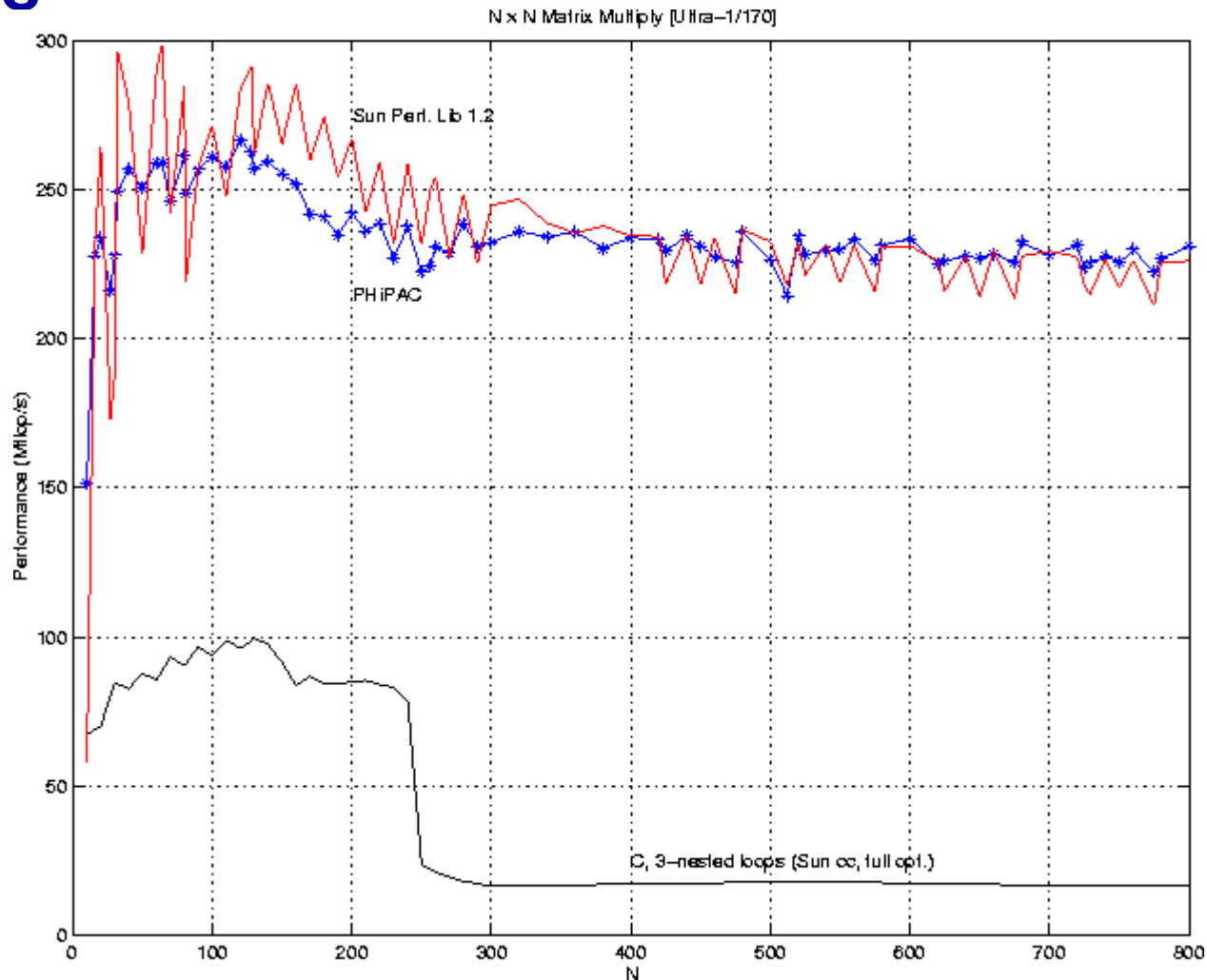
$$\text{So } q = f / m = 2n^3 / (n^3 + 3n^2)$$

$\approx 2$  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

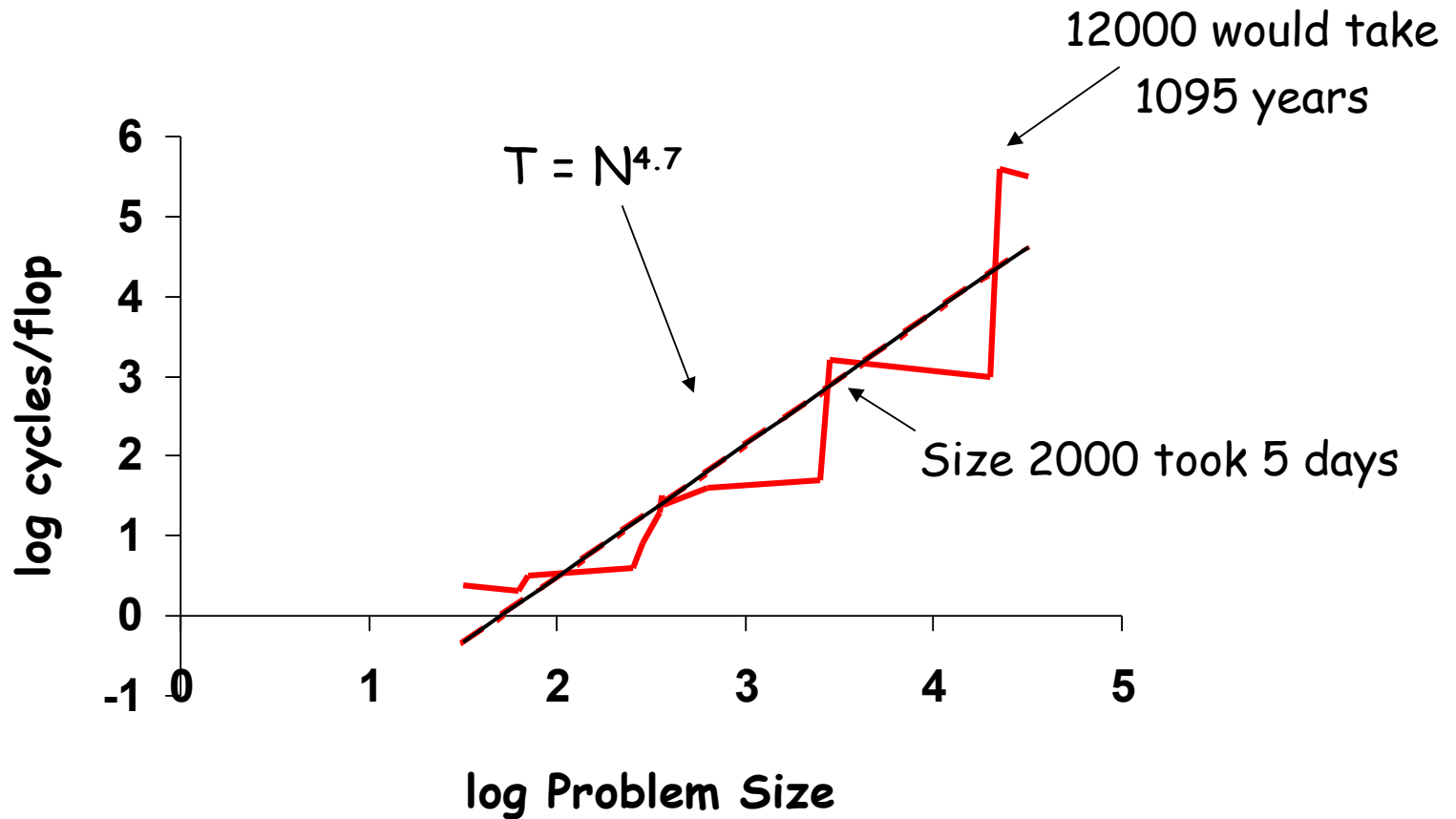


# Matrix-multiply, optimized several ways



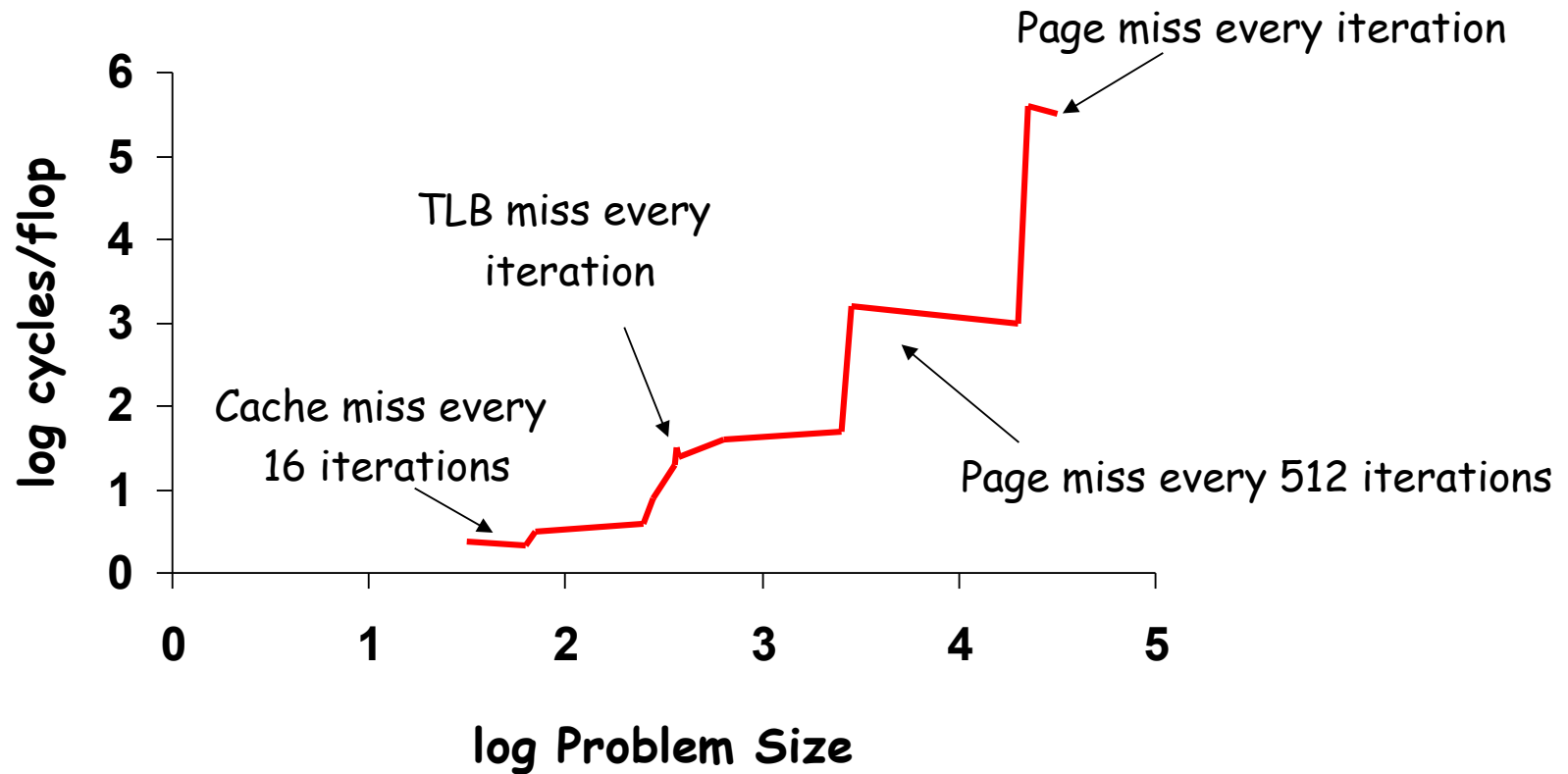
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

# Naïve Matrix Multiply on RS/6000



$O(N^3)$  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}

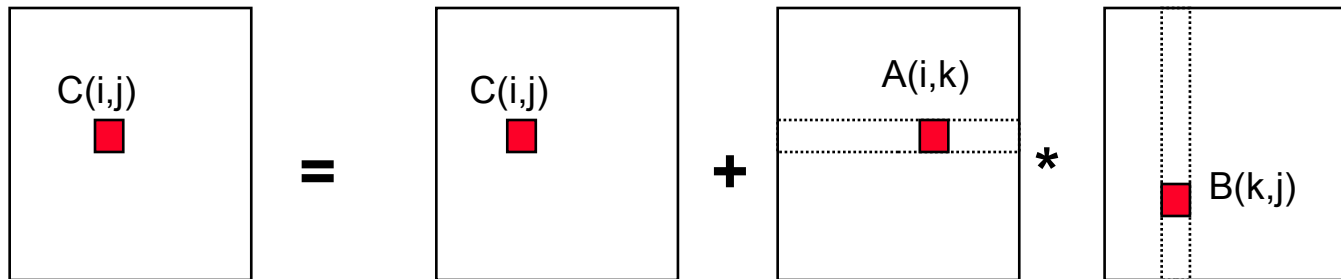
for  $k = 1$  to  $N$

{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}



# Blocked (Tiled) Matrix Multiply

---

Recall:

$m$  is amount memory traffic between slow and fast memory

matrix has  $n \times n$  elements, and  $N \times N$  blocks each of size  $b \times b$

$f$  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:

$$\begin{aligned} m &= N * n^2 && \text{read each block of B } N^3 \text{ times } (N^3 * b^2 = N^3 * (n/N)^2 = N * n^2) \\ &+ N * n^2 && \text{read each block of A } N^3 \text{ times} \\ &+ 2n^2 && \text{read and write each block of C once} \\ &= (2N + 2) * n^2 \end{aligned}$$

$$\begin{aligned} \text{So computational intensity } q &= f / m = 2n^3 / ((2N + 2) * n^2) \\ &\approx n / N = b \text{ for large } n \end{aligned}$$

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  $M_{\text{fast}}$

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

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

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

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

	<b>t_m/t_f</b>	<b>required KB</b>
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:


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

- There is a lower bound result that says we cannot do any better than this (using only associativity)
- **Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to  $q = O( (M_{\text{fast}})^{1/2} )$** 
  - #words moved between fast and slow memory =  $\Omega( n^3 / (M_{\text{fast}})^{1/2} )$



# **Review of lecture 2 so far (and a look ahead)**

**Layers**

- 
- **Applications**
    - How to decompose into well-understood algorithms (and their implementations)
  - **Algorithms (matmul as example)**
    - Need simple model of hardware to guide design, analysis: minimize accesses to slow memory
    - If lucky, theory describing “best algorithm”
  - **Software tools**
    - How do I implement my applications and algorithms in most efficient and productive way?
  - **Hardware**
    - Even simple programs have complicated behaviors
    - “Small” changes make execution time vary by orders of magnitude

# Communication lower bounds for Matmul

---

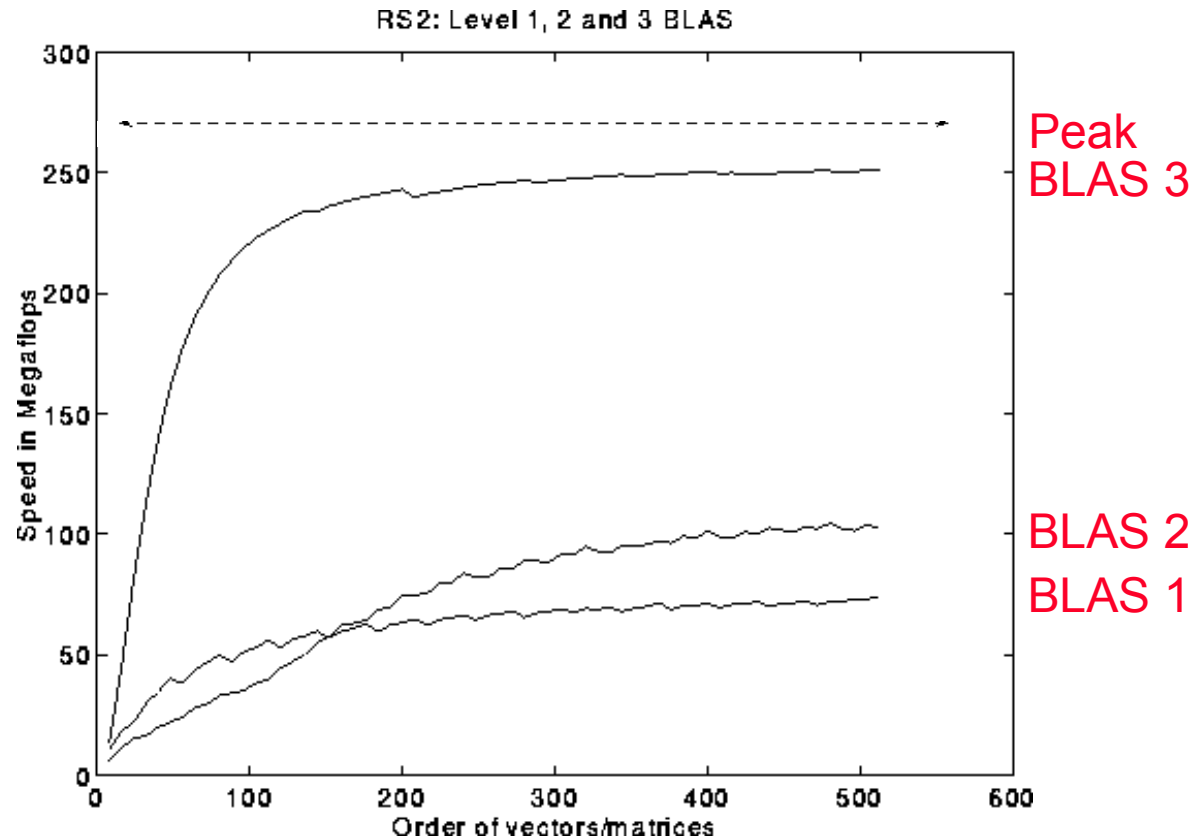
- Hong/Kung theorem is a lower bound on amount of data communicated by matmul
  - Number of words moved between fast and slow memory (cache and DRAM, or DRAM and disk, or ...) =  $\Omega(n^3 / M_{\text{fast}}^{1/2})$
- Cost of moving data may also depend on the number of “messages” into which data is packed
  - Eg: number of cache lines, disk accesses, ...
  - #messages =  $\Omega(n^3 / M_{\text{fast}}^{3/2})$
- 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, ...
- Need (more) new algorithms to attain these lower bounds...

# Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
  - [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/blas/blast--forum](http://www.netlib.org/blas/blast--forum)
- Vendors, others supply optimized implementations
- History
  - **BLAS1 (1970s):**
    - vector operations: dot product, saxpy ( $y = \alpha * x + y$ ), etc
    - $m = 2 * n$ ,  $f = 2 * n$ ,  $q = f / m = \text{computational intensity} \sim 1$  or less
  - **BLAS2 (mid 1980s)**
    - matrix-vector operations: matrix vector multiply, etc
    - $m = n^2$ ,  $f = 2 * n^2$ ,  $q \sim 2$ , less overhead
    - somewhat faster than BLAS1
  - **BLAS3 (late 1980s)**
    - matrix-matrix operations: matrix matrix multiply, etc
    - $m \leq 3n^2$ ,  $f = O(n^3)$ , so  $q = f / m$  can possibly be as large as  $n$ , so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK & ScaLAPACK)
  - See [www.netlib.org/{lapack,scalapack}](http://www.netlib.org/{lapack,scalapack})
  - More later in course

# BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

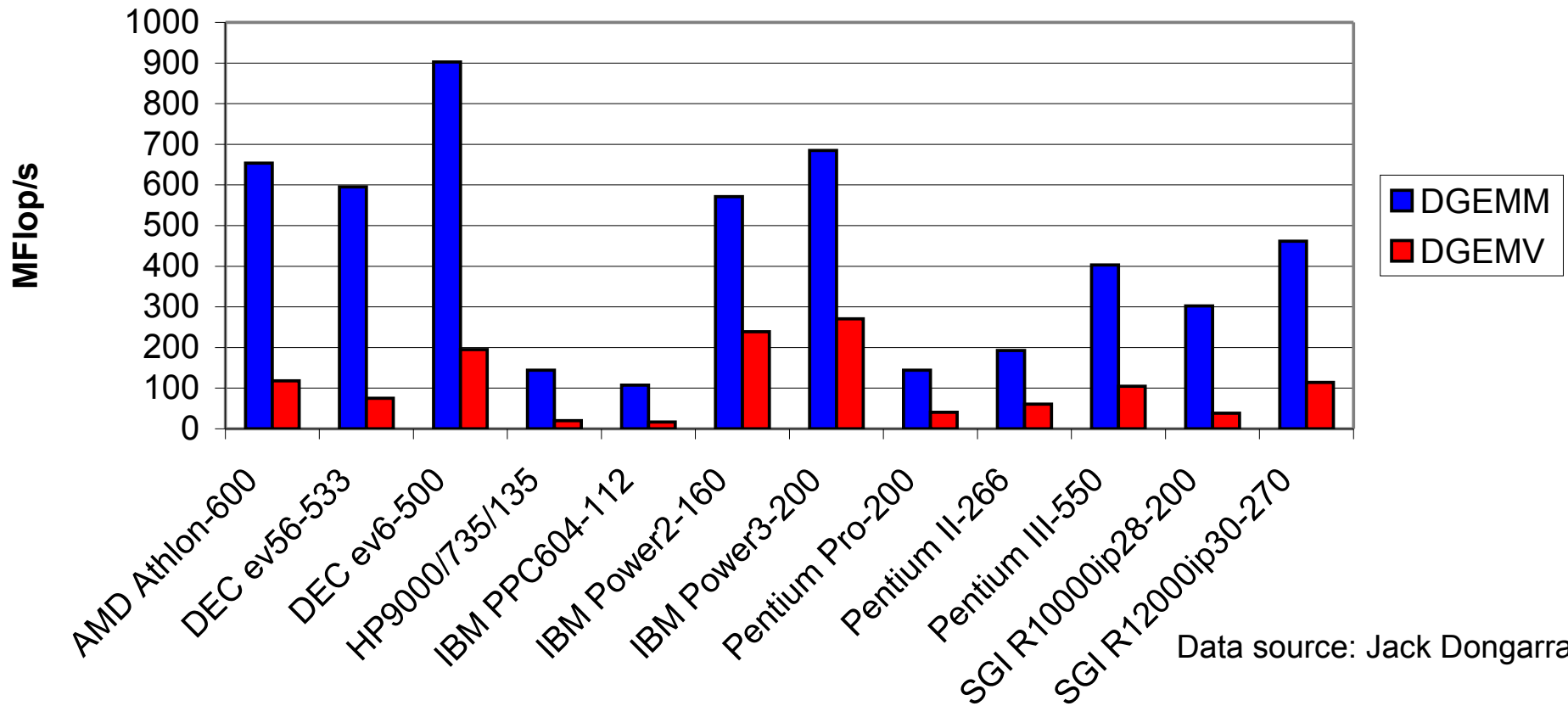


BLAS 3 (n-by-n matrix matrix multiply) vs  
BLAS 2 (n-by-n matrix vector multiply) vs  
BLAS 1 (saxpy of n vectors)

# Dense Linear Algebra: BLAS2 vs. BLAS3

- BLAS2 and BLAS3 have very different computational intensity, and therefore different performance

## BLAS3 (MatrixMatrix) vs. BLAS2 (MatrixVector)



# What if there are more than 2 levels of memory?

- Need to minimize communication between all levels
  - Between L1 and L2 cache, cache and DRAM, DRAM and disk...
- The tiled algorithm requires finding a good block size
  - Machine dependent
  - Need to “block”  $b \times b$  matrix multiply in inner most loop
    - 1 level of memory  $\Rightarrow$  3 nested loops (naïve algorithm)
    - 2 levels of memory  $\Rightarrow$  6 nested loops
    - 3 levels of memory  $\Rightarrow$  9 nested loops ...
- Cache Oblivious Algorithms offer an alternative
  - Treat  $n \times n$  matrix multiply as a set of smaller problems
  - Eventually, these will fit in cache
  - Will minimize # words moved between every level of memory hierarchy – at least asymptotically

# Recursive Matrix Multiplication (RMM) (1/2)

$$\begin{aligned} \bullet C &= \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix} \end{aligned}$$

- True when each  $A_{ij}$  etc  $1 \times 1$  or  $n/2 \times n/2$
- For simplicity: square matrices with  $n = 2^m$

```
func C = RMM (A, B, n)
  if n = 1, C = A * B, else
    { C11 = RMM (A11, B11, n/2) + RMM (A12, B21, n/2)
      C12 = RMM (A11, B12, n/2) + RMM (A12, B22, n/2)
      C21 = RMM (A21, B11, n/2) + RMM (A22, B21, n/2)
      C22 = RMM (A21, B12, n/2) + RMM (A22, B22, n/2) }
  return
```

## Recursive Matrix Multiplication (2/2)

```
func C = RMM (A, B, n)
  if n=1, C = A * B, else
    { C11 = RMM (A11, B11, n/2) + RMM (A12, B21, n/2)
      C12 = RMM (A11, B12, n/2) + RMM (A12, B22, n/2)
      C21 = RMM (A21, B11, n/2) + RMM (A22, B21, n/2)
      C22 = RMM (A21, B12, n/2) + RMM (A22, B22, n/2) }
  return
```

$A(n)$  = # arithmetic operations in  $\text{RMM}(\cdot, \cdot, n)$   
 $= 8 \cdot A(n/2) + 4(n/2)^2$  if  $n > 1$ , else 1  
 $= 2n^3$  ... same operations as usual, in different order

$M(n)$  = # words moved between fast, slow memory by  $\text{RMM}(\cdot, \cdot, n)$   
 $= 8 \cdot M(n/2) + 4(n/2)^2$  if  $3n^2 > M_{\text{fast}}$ , else  $3n^2$   
 $= O(n^3 / (M_{\text{fast}})^{1/2} + n^2)$  ... same as blocked matmul

Don't need to know  $M_{\text{fast}}$  for this to work!



# Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat  $n \times n$  matrix multiply set of smaller problems
  - Eventually, these will fit in cache
- Cases for  $A (n \times m) * B (m \times p)$ 
  - Case 1:  $m \geq \max\{n, p\}$ : split A horizontally:
  - Case 2 :  $n \geq \max\{m, p\}$ : split A vertically and B horizontally
  - Case 3:  $p \geq \max\{m, n\}$ : split B vertically

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

**Case 1**

$$(A_1, A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B_1 + A_2 B_2)$$

**Case 2**

$$A(B_1, B_2) = (A B_1, A B_2)$$

**Case 3**

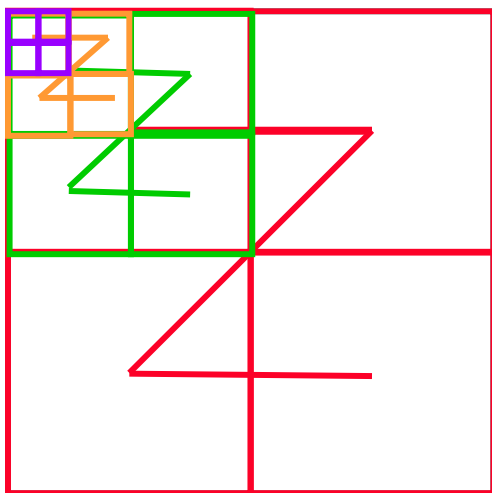
# **Experience with Cache-Oblivious Algorithms**

- In practice, need to cut off recursion well before 1x1 blocks
  - Call “micro-kernel” on small blocks
- Implementing a high-performance Cache-Oblivious code is not easy
  - Careful attention to micro-kernel is needed
- Using fully recursive approach with highly optimized recursive micro-kernel, Pingali et al report that they never got more than 2/3 of peak.
- Issues with Cache Oblivious (recursive) approach
  - Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Pre-fetching is needed to compete with best code: not well-understood in the context of Cache-Oblivious codes

**Unpublished work, presented at LACSI 2006**

# Recursive Data Layouts

- A related idea is to use a recursive structure for the matrix
  - Improve locality with machine-independent data structure
  - Can minimize latency with multiple levels of memory hierarchy
- There are several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering (“space filling curve”)
- See papers on “cache oblivious algorithms” and “recursive layouts”
  - Gustavson, Kagstrom, et al, SIAM Review, 2004



## Advantages:

- the recursive layout works well for any cache size

## Disadvantages:

- The index calculations to find  $A[i,j]$  are expensive
- Implementations switch to column-major for small sizes

# Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has  $O(n^3)$  flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

$$\text{Let } M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\text{Let } p1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$p5 = a_{11} * (b_{12} - b_{22})$$

$$p2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$p6 = a_{22} * (b_{21} - b_{11})$$

$$p3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

$$p7 = (a_{21} + a_{22}) * b_{11}$$

$$p4 = (a_{11} + a_{12}) * b_{22}$$

$$\text{Then } m_{11} = p1 + p2 - p4 + p6$$

$$m_{12} = p4 + p5$$

$$m_{21} = p6 + p7$$

$$m_{22} = p2 - p3 + p5 - p7$$

Extends to nxn by divide&conquer

## Strassen (continued)

---

$$\begin{aligned} T(n) &= \text{Cost of multiplying } n \times n \text{ matrices} \\ &= 7 * T(n/2) + 18 * (n/2)^2 \\ &= O(n \log_2 7) \\ &= O(n^{2.81}) \end{aligned}$$

- Asymptotically faster
  - Several times faster for large  $n$  in practice
  - Cross-over depends on machine
  - “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98
- Possible to extend communication lower bound to Strassen
  - #words moved between fast and slow memory  
 $= \Omega(n^{\log_2 7} / M^{(\log_2 7)/2 - 1}) \sim \Omega(n^{2.81} / M^{0.4})$  (Ballard, D., Holtz, Schwartz, 2011)
  - Attainable too

# Other Fast Matrix Multiplication Algorithms

- World's record was  $O(n^{2.376...})$ 
  - Coppersmith & Winograd, 1987
- New Record! 2.376 reduced to 2.373
  - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Lower bound on #words moved can be extended to (some) of these algorithms
- Possibility of  $O(n^{2+\epsilon})$  algorithm!
  - Cohn, Umans, Kleinberg, 2003
- Can show they all can be made numerically stable
  - D., Dumitriu, Holtz, Kleinberg, 2007
- Can do rest of linear algebra (solve  $Ax=b$ ,  $Ax=\lambda x$ , etc) as fast, and numerically stably
  - D., Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need

# Tuning Code in Practice

---

- Tuning code can be tedious
  - Lots of code variations to try besides blocking
  - Machine hardware performance hard to predict
  - Compiler behavior hard to predict
- Response: “Autotuning”
  - Let computer generate large set of possible code variations, and search them for the fastest ones
  - Field started with CS267 homework assignment in mid 1990s
    - PHiPAC, leading to ATLAS, incorporated in Matlab
    - We still use the same assignment
  - We (and others) are extending autotuning to other dwarfs / motifs
- Still need to understand how to do it by hand
  - Not every code will have an autotuner
  - Need to know if you want to build autotuners

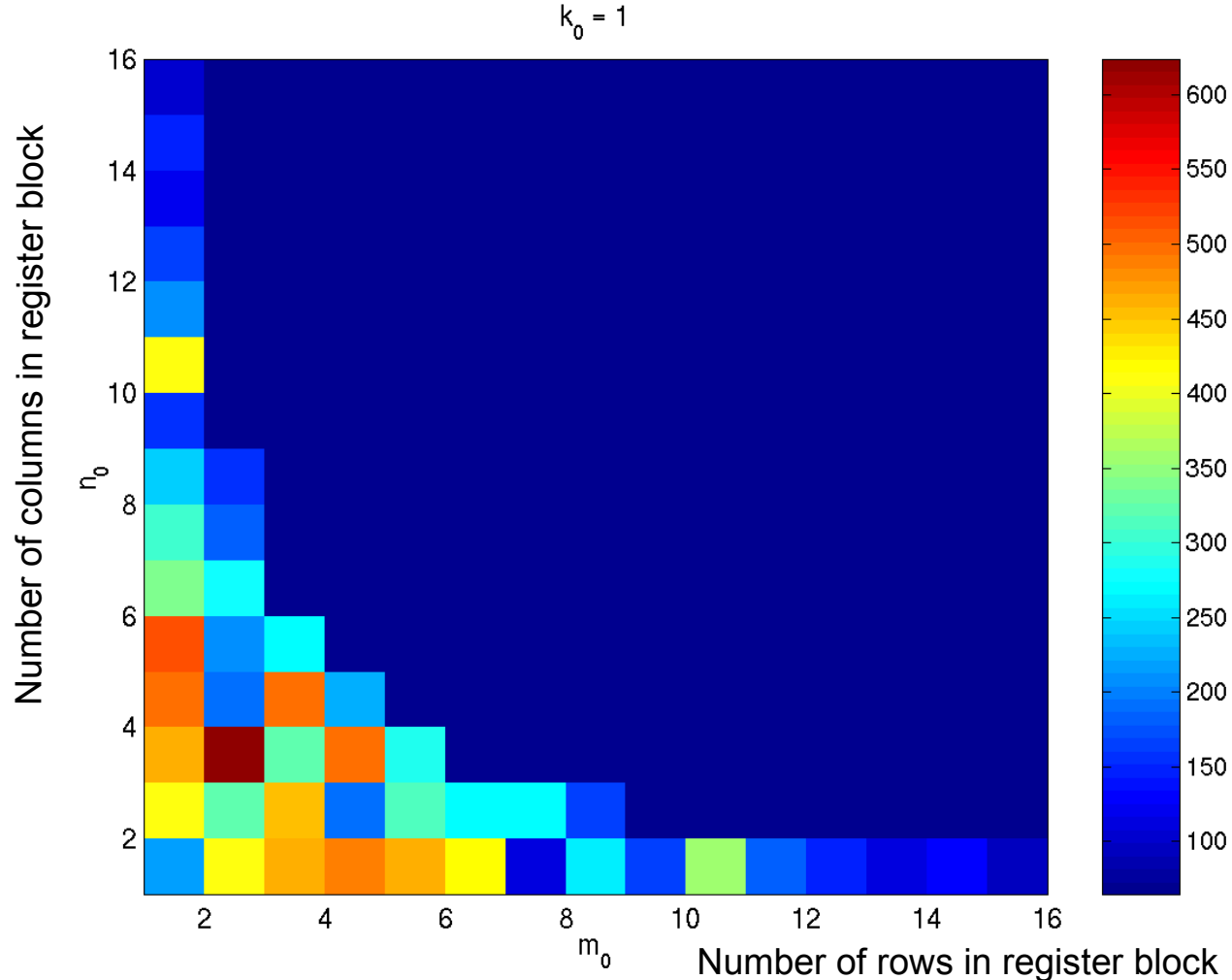
# Search Over Block Sizes

---

- Performance models are useful for high level algorithms
  - Helps in developing a blocked algorithm
  - Models have not proven very useful for block size selection
    - too complicated to be useful
      - See work by Sid Chatterjee for detailed model
    - too simple to be accurate
      - Multiple multidimensional arrays, virtual memory, etc.
  - Speed depends on matrix dimensions, details of code, compiler, processor



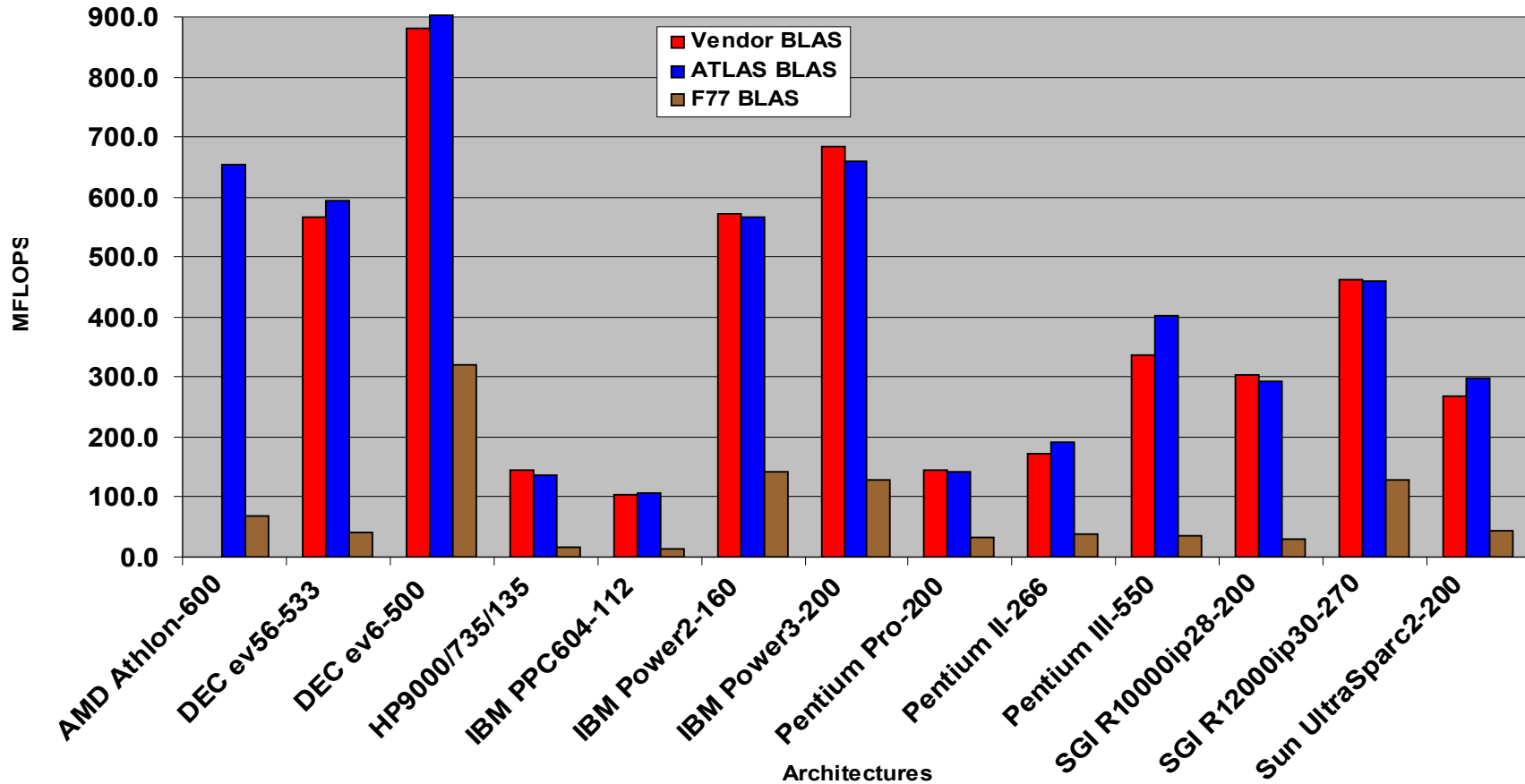
# What the Search Space Looks Like



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.  
(Platform: Sun Ultra-Ili, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

# ATLAS (DGEMM $n = 500$ )

Source: Jack Dongarra



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

# Optimizing in Practice

---

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions
- Hard to do by hand (but you’ll try)
- Automatic optimization an active research area
  - ParLab: [parlab.eecs.berkeley.edu](http://parlab.eecs.berkeley.edu)
  - BeBOP: [bebop.cs.berkeley.edu](http://bebop.cs.berkeley.edu)
    - Weekly group meeting Tuesdays 12:30pm
  - PHiPAC: [www.icsi.berkeley.edu/~bilmes/hipac](http://www.icsi.berkeley.edu/~bilmes/hipac)  
in particular [tr-98-035.ps.gz](http://tr-98-035.ps.gz)
  - ATLAS: [www.netlib.org/atlas](http://www.netlib.org/atlas)

# Removing False Dependencies

---

- Using local variables, reorder operations to remove false dependencies

```
a[i] = b[i] + c;  
a[i+1] = b[i+1] * d;
```

false read-after-write hazard  
between a[i] and b[i+1]



```
float f1 = b[i];  
float f2 = b[i+1];
```

```
a[i] = f1 + c;  
a[i+1] = f2 * d;
```

With some compilers, you can declare a and b unaliased.

- Done via “restrict pointers,” compiler flag, or pragma)

# Exploit Multiple Registers

---

- Reduce demands on memory bandwidth by pre-loading into local variables

```
while( ... ) {  
    *res++ = filter[0]*signal[0]  
            + filter[1]*signal[1]  
            + filter[2]*signal[2];  
    signal++;  
}
```



```
float f0 = filter[0];  
float f1 = filter[1];  
float f2 = filter[2];  
while( ... ) {  
    *res++ = f0*signal[0]  
            + f1*signal[1]  
            + f2*signal[2];  
    signal++;  
}
```

also: register float f0 = ...;

Example is a convolution

# Loop Unrolling

---

- Expose instruction-level parallelism

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
    signal += 3;
    s0 = signal[0];
    res[0] = f0*s1 + f1*s2 + f2*s0;

    s1 = signal[1];
    res[1] = f0*s2 + f1*s0 + f2*s1;

    s2 = signal[2];
    res[2] = f0*s0 + f1*s1 + f2*s2;

    res += 3;
} while( ... );
```

# Expose Independent Operations

---

- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

```
f1 = f5 * f9;  
f2 = f6 + f10;  
f3 = f7 * f11;  
f4 = f8 + f12;
```

# Copy optimization

---

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: use different data structures from start (if users willing)
    - Recall recursive data layouts

Original matrix  
(numbers are addresses)



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

Reorganized into  
2x2 blocks

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



# Locality in Other Algorithms

---

- The performance of any algorithm is limited by  $q$ 
  - $q = \text{“computational intensity”} = \# \text{arithmetic\_ops} / \# \text{words\_moved}$
- In matrix multiply, we increase  $q$  by changing computation order
  - increased temporal locality
- For other algorithms and data structures, even hand-transformations are still an open problem
  - Lots of open problems, class projects

# 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 that recently used
- 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 Feb 2
  - Could even be your intended class project
- Homework 1 posted on web site
  - Tuning matrix multiply
  - Due Feb 14
- Please fill in on-line class survey
  - We need this to assign teams for Homework 1

## Some reading for today (see website)

---

- Sourcebook Chapter 3, (note that chapters 2 and 3 cover the material of lecture 2 and lecture 3, but not in the same order).
- "[Performance Optimization of Numerically Intensive Codes](#)", by Stefan Goedecker and Adolfo Hoisie, SIAM 2001.
- Web pages for reference:
  - [BeBOP Homepage](#)
  - [ATLAS Homepage](#)
  - [BLAS](#) (Basic Linear Algebra Subroutines), Reference for (unoptimized) implementations of the BLAS, with documentation.
  - [LAPACK](#) (Linear Algebra PACKage), a standard linear algebra library optimized to use the BLAS effectively on uniprocessors and shared memory machines (software, documentation and reports)
  - [ScaLAPACK](#) (Scalable LAPACK), a parallel version of LAPACK for distributed memory machines (software, documentation and reports)
- Tuning Strassen's Matrix Multiplication for Memory Efficiency  
Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck  
in Proceedings of Supercomputing '98, November 1998 [postscript](#)
- Recursive Array Layouts and Fast Parallel Matrix Multiplication" by Chatterjee et al. IEEE TPDS November 2002.

---

## Extra Slides

# Memory Performance on Itanium 2 (CITRIS)

Itanium2, 900 MHz

MAPS Loads [Itanium2-linux-ecc7]

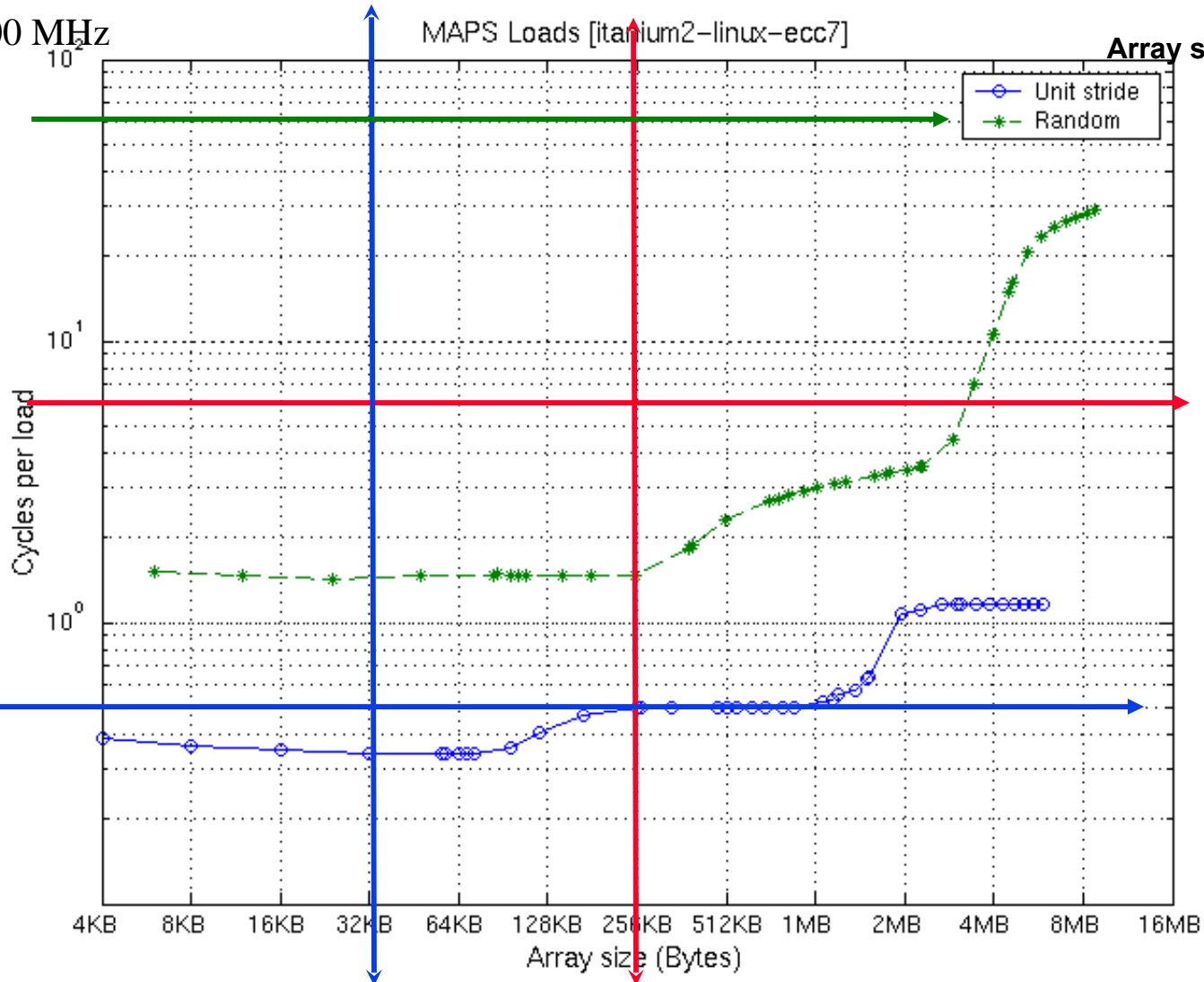
Array size

**Mem:**  
11-60 cycles

**L3: 2 MB**  
128 B line  
3-20 cycles

**L2: 256 KB**  
128 B line  
.5-4 cycles

**L1: 32 KB**  
64B line  
.34-1 cycles



Uses MAPS Benchmark: <http://www.sdsc.edu/PMaC/MAPs/maps.html>

# Proof of Communication Lower Bound on $C = A*B$ (1/6)

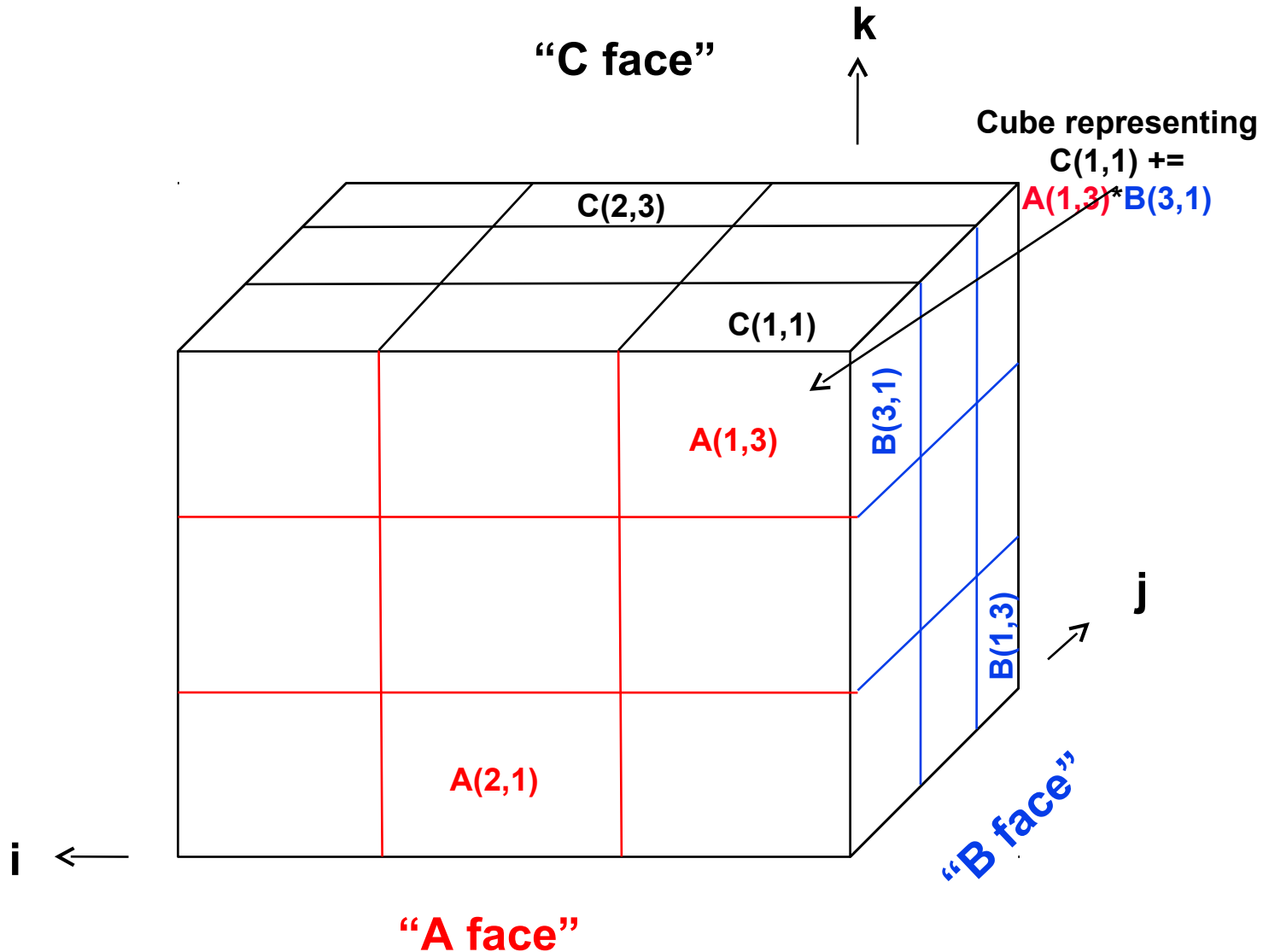
- Proof from Irony/Tishkin/Toledo (2004)
  - We'll need it for the communication lower bound on parallel matmul
- Think of instruction stream being executed
  - Looks like “ ... add, load, multiply, store, load, add, ... ”
  - We want to count the number of loads and stores, given that we are multiplying n-by-n matrices  $C = A*B$  using the usual  $2 \cdot n^3$  flops, possibly reordered assuming addition is commutative/associative
    - It actually isn't associative in floating point, but close enough
  - Assuming that at most M words can be stored in fast memory
- Outline:
  - Break instruction stream into segments, each containing M loads and stores
  - Somehow bound the maximum number of adds and multiplies that could be done in each segment, call it F
  - So  $F \cdot \# \text{ segments} \geq 2 \cdot n^3$ , and  $\# \text{ segments} \geq 2 \cdot n^3 / F$
  - So  $\# \text{ loads \& stores} = M \cdot \# \text{ segments} \geq M \cdot 2 \cdot n^3 / F$

## Proof of Communication Lower Bound on $C = A * B$ (2/6)

- Given segment of instruction stream with  $M$  loads & stores, how many adds & multiplies ( $F$ ) can we do?
  - At most  $2M$  entries of  $C$ ,  $2M$  entries of  $A$  and/or  $2M$  entries of  $B$  can be accessed
- Use geometry:
  - Represent  $2 \cdot n^3$  operations by  $n \times n \times n$  cube
  - One  $n \times n$  face represents  $A$ 
    - each  $1 \times 1$  subsquare represents one  $A(i,k)$
  - One  $n \times n$  face represents  $B$ 
    - each  $1 \times 1$  subsquare represents one  $B(k,j)$
  - One  $n \times n$  face represents  $C$ 
    - each  $1 \times 1$  subsquare represents one  $C(i,j)$
  - Each  $1 \times 1 \times 1$  subcube represents one  $C(i,j) += A(i,k) * B(k,j)$



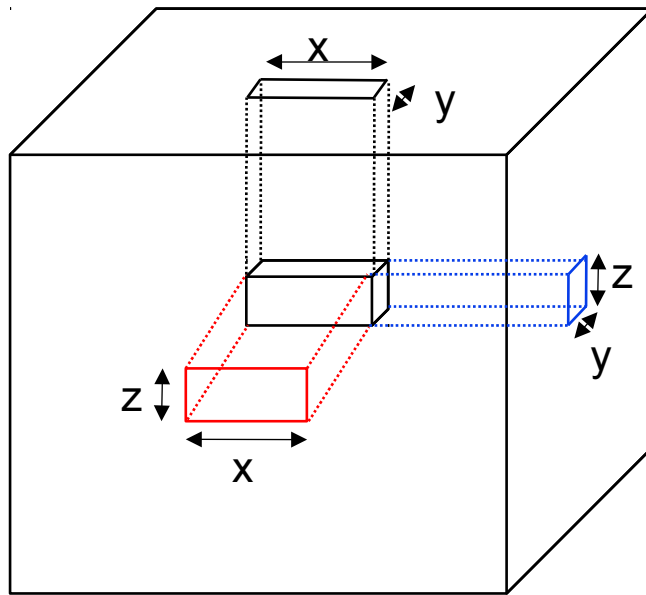
# Proof of Communication Lower Bound on $C = A * B$ (3/6)



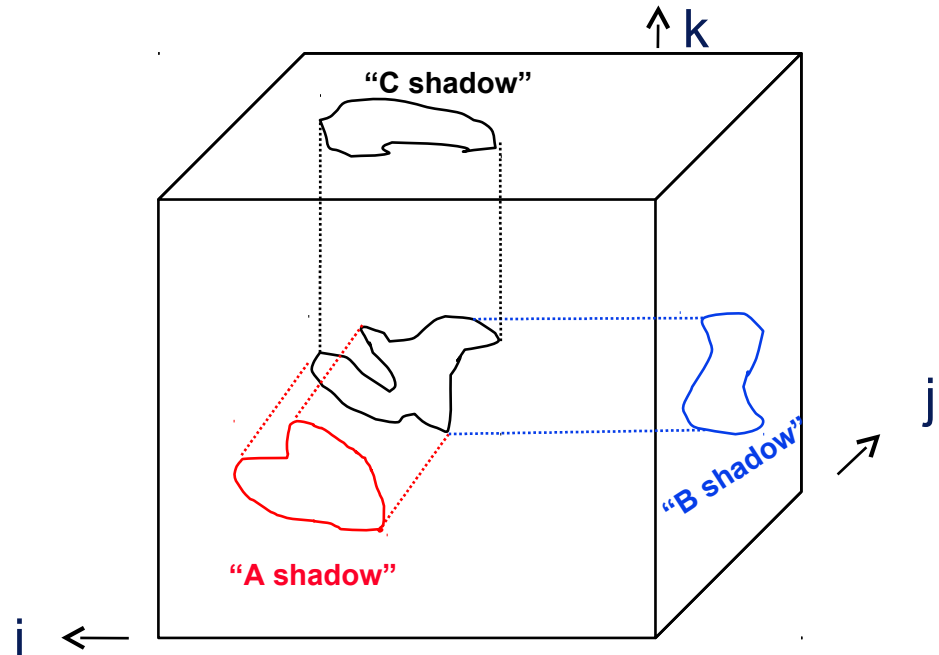
## Proof of Communication Lower Bound on $C = A * B$ (4/6)

- Given segment of instruction stream with  $M$  load & stores, how many adds & multiplies ( $F$ ) can we do?
  - At most  $2M$  entries of  $C$ , and/or of  $A$  and/or of  $B$  can be accessed
- Use geometry:
  - Represent  $2 \cdot n^3$  operations by  $n \times n \times n$  cube
  - One  $n \times n$  face represents  $A$ 
    - each  $1 \times 1$  subsquare represents one  $A(i,k)$
  - One  $n \times n$  face represents  $B$ 
    - each  $1 \times 1$  subsquare represents one  $B(k,j)$
  - One  $n \times n$  face represents  $C$ 
    - each  $1 \times 1$  subsquare represents one  $C(i,j)$
  - Each  $1 \times 1 \times 1$  subcube represents one  $C(i,j) += A(i,k) * B(k,j)$
- If we have at most  $2M$  “A squares”,  $2M$  “B squares”, and  $2M$  “C squares” on faces, how many cubes can we have?

# Proof of Communication Lower Bound on $C = A*B$ (5/6)



# cubes in black box with  
side lengths  $x$ ,  $y$  and  $z$   
= Volume of black box  
=  $x*y*z$   
=  $(\#A_{\square s} * \#B_{\square s} * \#C_{\square s})^{1/2}$   
=  $(xz * zy * yx)^{1/2}$



$(i,k)$  is in “A shadow” if  $(i,j,k)$  in 3D set  
 $(j,k)$  is in “B shadow” if  $(i,j,k)$  in 3D set  
 $(i,j)$  is in “C shadow” if  $(i,j,k)$  in 3D set

Thm (Loomis & Whitney, 1949)

# cubes in 3D set = Volume of 3D set  
 $\leq (\text{area(A shadow)} * \text{area(B shadow)} * \text{area(C shadow)})^{1/2}$

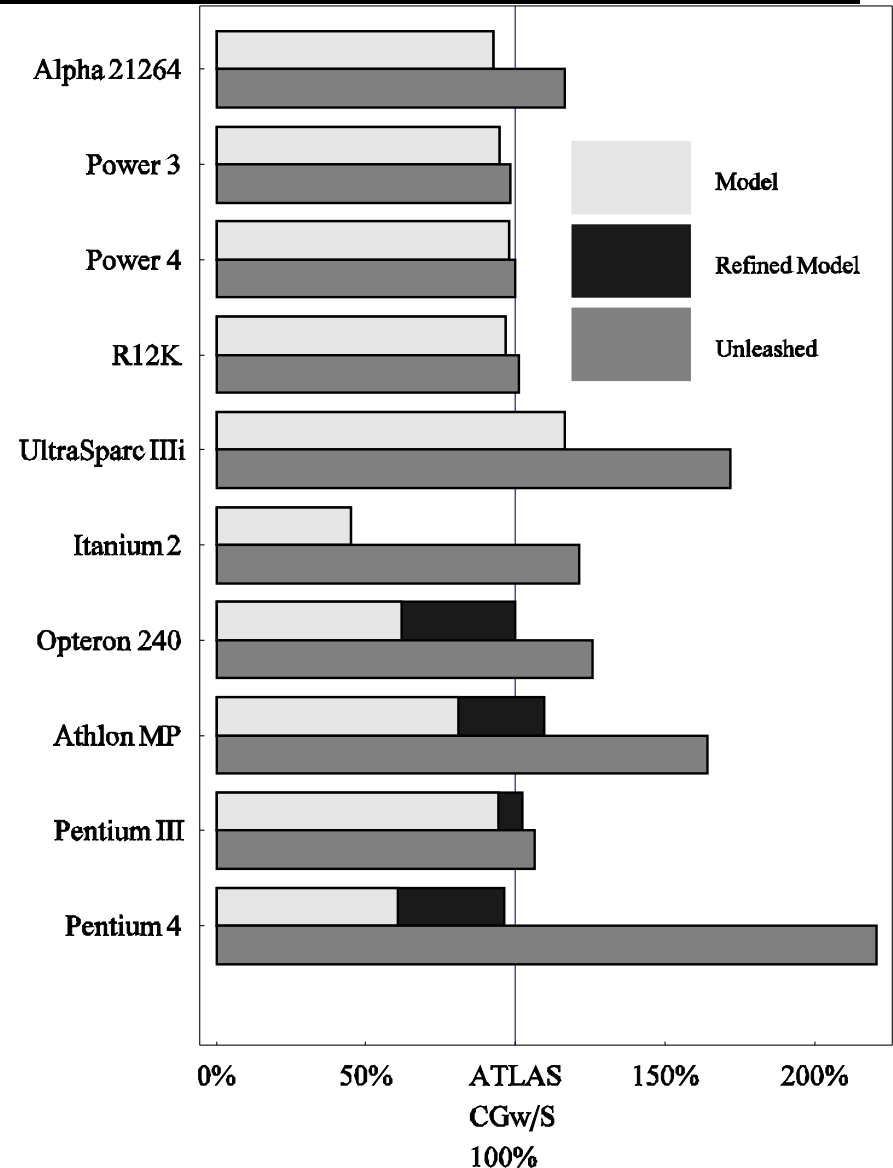
## Proof of Communication Lower Bound on $C = A*B$ (6/6)

- Consider one “segment” of instructions with  $M$  loads and stores
- There can be at most  $2M$  entries of  $A$ ,  $B$ ,  $C$  available in one segment
- Volume of set of cubes representing possible multiply/adds  $\leq (2M \cdot 2M \cdot 2M)^{1/2} = (2M)^{3/2} \equiv F$
- # Segments  $\geq 2n^3 / F$
- # Loads & Stores =  $M \cdot \text{\#Segments} \geq M \cdot 2n^3 / F = n^3 / (2M)^{1/2}$
- Parallel Case: apply reasoning to one processor out of  $P$ 
  - # Adds and Muls =  $2n^3 / P$  (assuming load balanced)
  - $M = n^2 / P$  (each processor gets equal fraction of matrix)
  - # “Load & Stores” = # words communicated with other procs  $\geq M \cdot (2n^3 / P) / F = M \cdot (2n^3 / P) / (2M)^{3/2} = n^2 / (2P)^{1/2}$

# Experiments on Search vs. Modeling

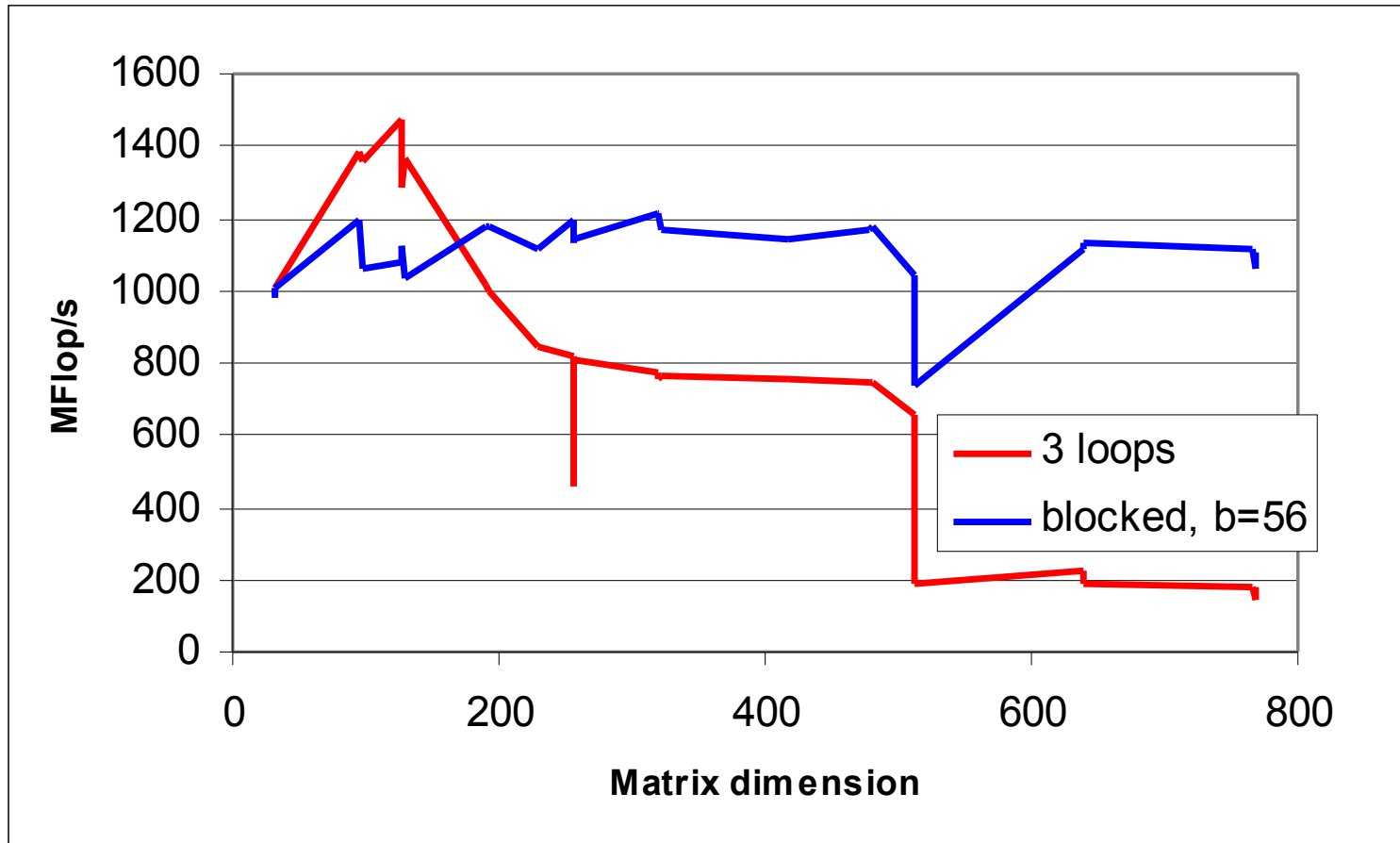
**Study compares search (Atlas) to optimization selection based on performance models**

- Ten modern architectures
- Model did well on most cases
  - Better on UltraSparc
  - Worse on Itanium
- Eliminating performance gaps: think globally, search locally
  - small performance gaps: local search
  - large performance gaps: refine model
- Substantial gap between ATLAS CGw/S and ATLAS Unleashed on some machines



# Tiling Alone Might Not Be Enough

- Naïve and a “naïvely tiled” code on Itanium 2
  - Searched all block sizes to find best,  $b=56$
  - Starting point for next homework



# Minimize Pointer Updates

---

- Replace pointer updates for strided memory addressing with constant array offsets

```
f0 = *r8; r8 += 4;  
f1 = *r8; r8 += 4;  
f2 = *r8; r8 += 4;
```



```
f0 = r8[0];  
f1 = r8[4];  
f2 = r8[8];  
r8 += 12;
```

Pointer vs. array expression costs may differ.

- Some compilers do a better job at analyzing one than the other

# Summary

---

- Performance programming on uniprocessors requires
  - understanding of memory system
  - understanding of fine-grained parallelism in processor
- Simple performance models can aid in understanding
  - Two ratios are key to efficiency (relative to peak)
    - 1.computational intensity of the algorithm:
      - $q = f/m = \# \text{ floating point operations} / \# \text{ slow memory references}$
    - 1.machine balance in the memory system:
      - $t_m/t_f = \text{time for slow memory reference} / \text{time for floating point operation}$
- Want  $q > t_m/t_f$  to get half machine peak
- Blocking (tiling) is a basic approach to increase  $q$ 
  - Techniques apply generally, but the details (e.g., block size) are architecture dependent
  - Similar techniques are possible on other data structures and algorithms
- Now it's your turn: Homework 1
  - 1.Work in teams of 2 or 3 (assigned this time)



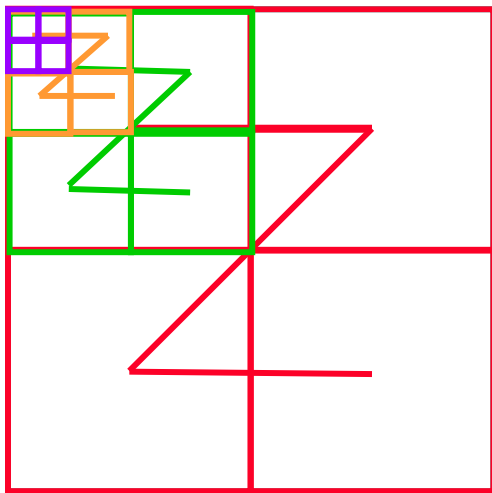
# Questions You Should Be Able to Answer

---

1. What is the key to understand algorithm efficiency in our simple memory model?
2. What is the key to understand machine efficiency in our simple memory model?
3. What is tiling?
4. Why does block matrix multiply reduce the number of memory references?
5. What are the BLAS?
6. Why does loop unrolling improve uniprocessor performance?

# Recursive Data Layouts

- Blocking seems to require knowing cache sizes – portable?
- A related (recent) idea is to use a recursive structure for the matrix
- There are several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering (“space filling curve”)
- See papers on “cache oblivious algorithms” and “recursive layouts”



## Advantages:

- the recursive layout works well for any cache size

## Disadvantages:

- The index calculations to find  $A[i,j]$  are expensive
- Implementations switch to column-major for small sizes