
Uniprocessor Optimizations and Matrix Multiplication

BeBOP Summer 2002

<http://www.cs.berkeley.edu/~richie/bebop>

Applications ...

- Scientific simulation and modeling
 - Weather and earthquakes
 - Cars and buildings
 - The universe
- Signal processing
 - Audio and image compression
 - Machine vision
 - Speech recognition
- Information retrieval
 - Web searching
 - Human genome
- Computer graphics and computational geometry
 - Structural models
 - Films: Final Fantasy, Shrek

... and their Building Blocks (Kernels)

- Scientific simulation and modeling
 - Matrix-vector/matrix-matrix multiply
 - Solving linear systems
- Signal processing
 - Performing fast transforms: Fourier, trigonometric, wavelet
 - Filtering
 - Linear algebra on structured matrices
- Information retrieval
 - Sorting
 - Finding eigenvalues and eigenvectors
- Computer graphics and computational geometry
 - Matrix multiply
 - Computing matrix determinants

Outline

- Parallelism in Modern Processors
- Memory Hierarchies
- Matrix Multiply Cache Optimizations
- Bag of Tricks

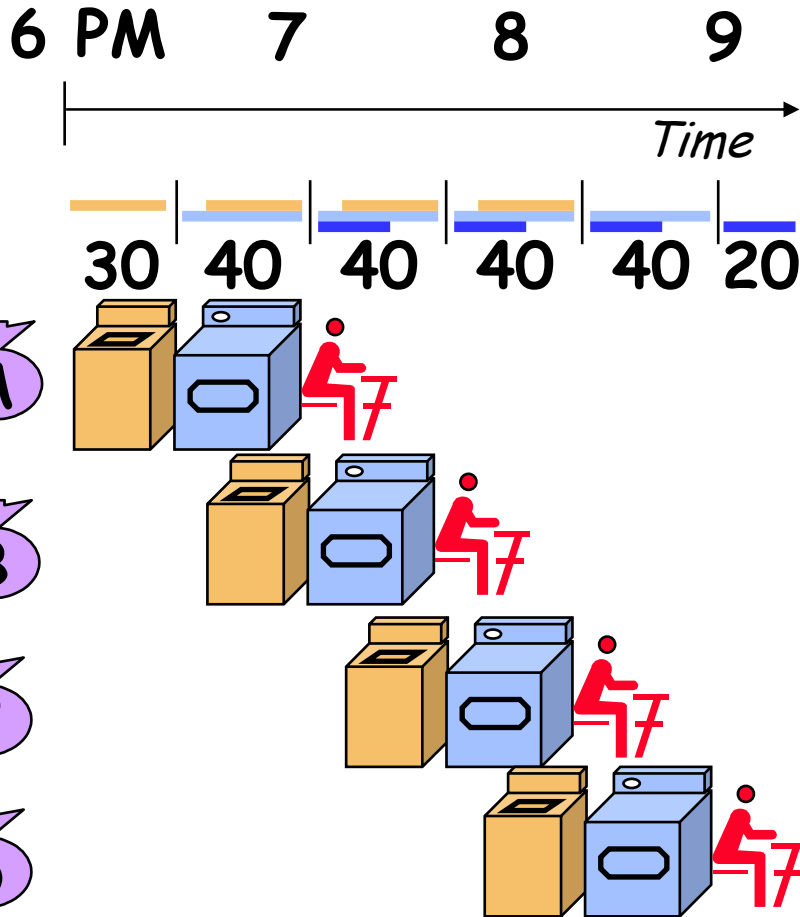
Modern Processors: Theory & Practice

- Idealized Uniprocessor Model
 - Execution order specified by program
 - Operations (load/store, +/*, branch) have roughly the same cost
- Processors in the Real World
 - Registers and caches
 - Small amounts of fast memory
 - Memory ops have widely varying costs
 - Exploit Instruction-Level Parallelism (ILP)
 - Superscalar — multiple functional units
 - Pipelined — decompose units of execution into parallel stages
 - Different instruction mixes/orders have different costs
- Why is this your problem?
 - In theory, compilers understand all this mumbo-jumbo and optimize your programs; in practice, they don't.

What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry

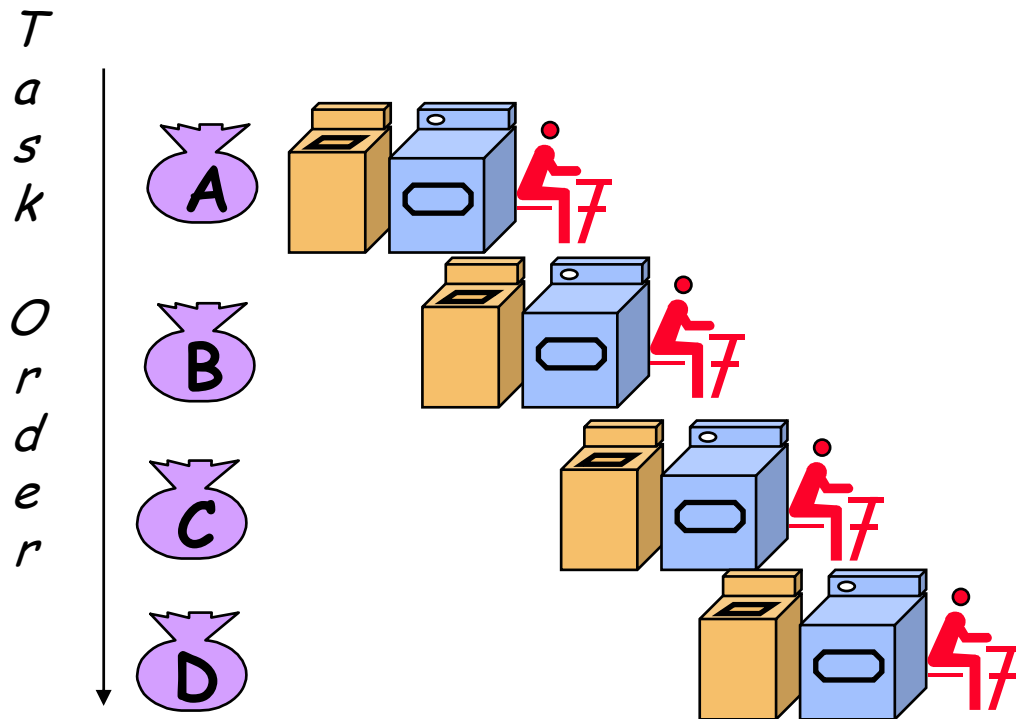
wash (30 min) + dry (40 min) + fold (20 min)



- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6$ hours
 - Pipelined execution takes $30 + 4 * 40 + 20 = 3.3$ hours
- Pipelining helps **throughput**, but not **latency**
- Pipeline rate limited by **slowest** pipeline stage
- Potential speedup = **Number pipe stages**
- Time to “fill” pipeline and time to “drain” it reduces speedup

Limits of ILP

Hazards prevent next instruction from executing in its designated clock cycle



- **Structural**: single person to fold and put clothes away
- **Data**: missing socks
- **Control**: dyed clothes need to be rewashed

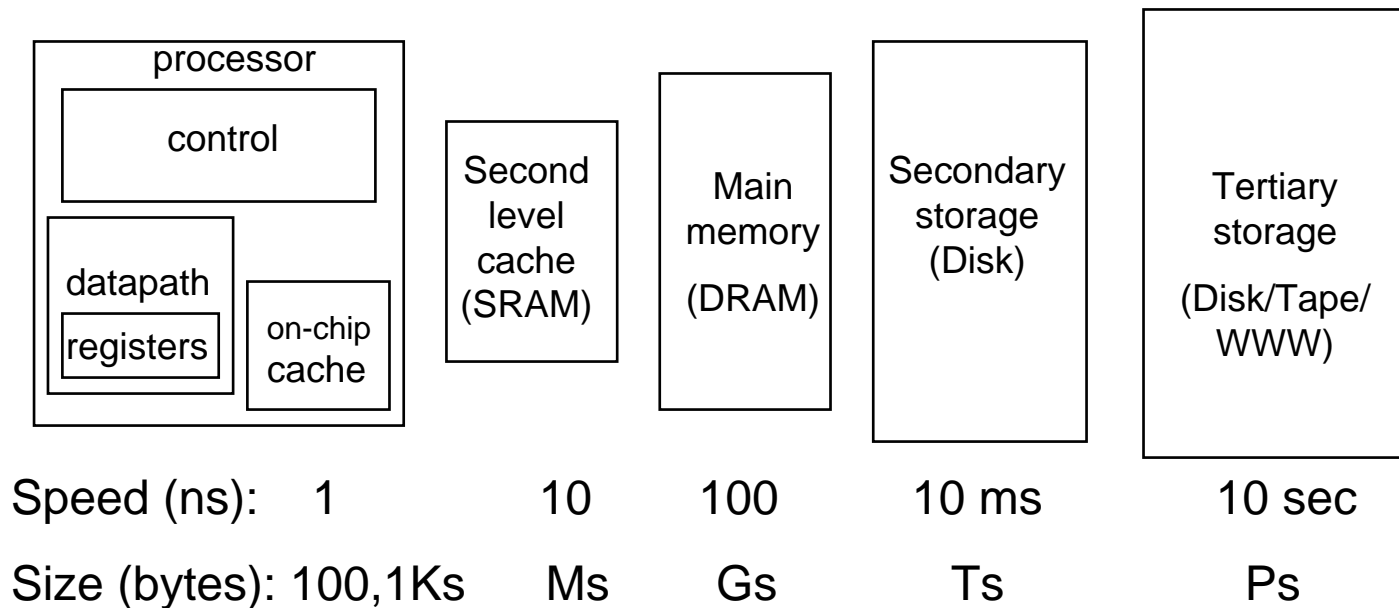
- Compiler will try to reduce these, but careful coding helps!

Outline

- Parallelism in Modern Processors
- Memory Hierarchies
- Matrix Multiply Cache Optimizations
- Bag of Tricks

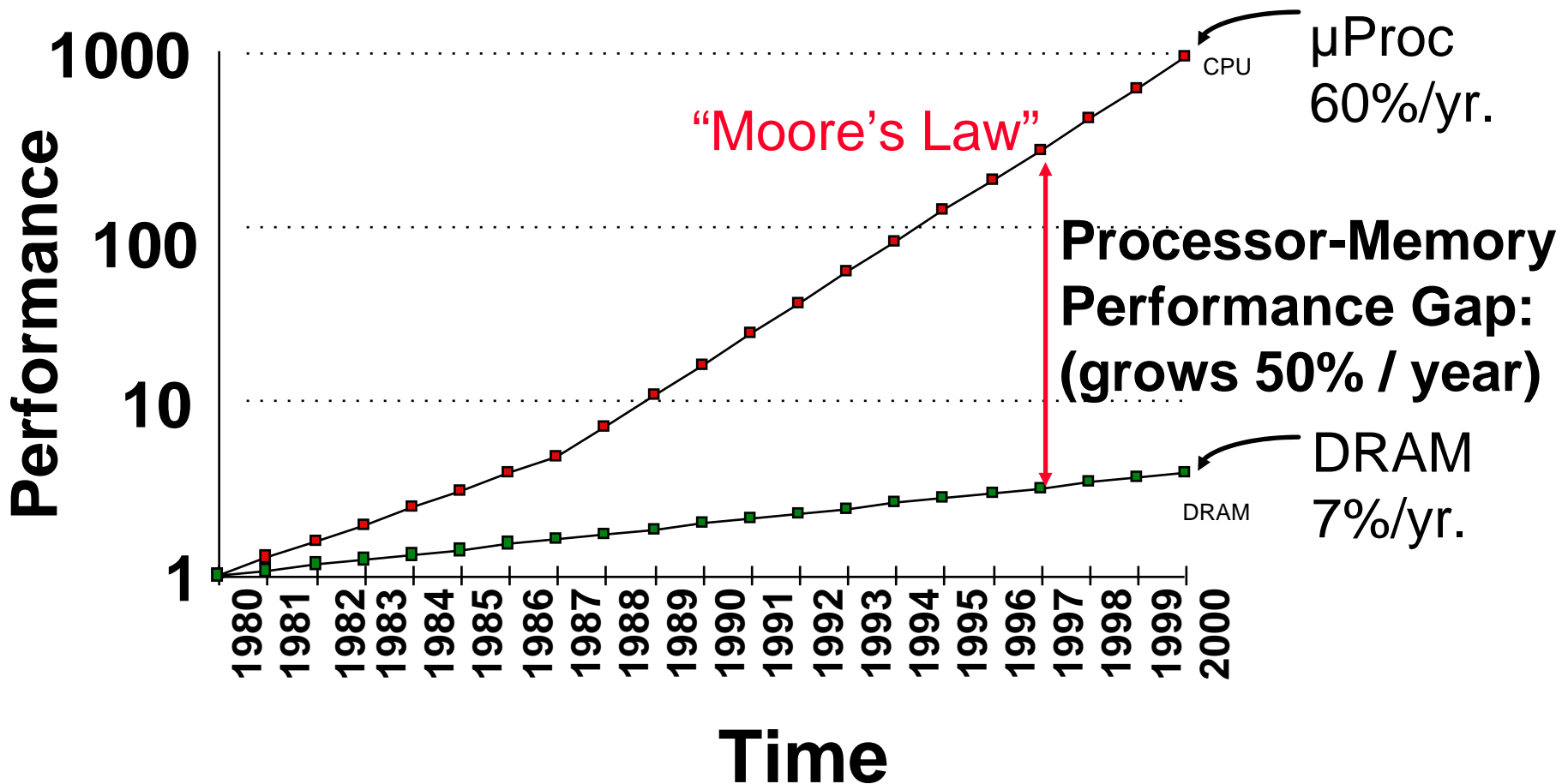
Memory Hierarchy

- 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



Processor-DRAM Gap (latency)

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



Cache Basics

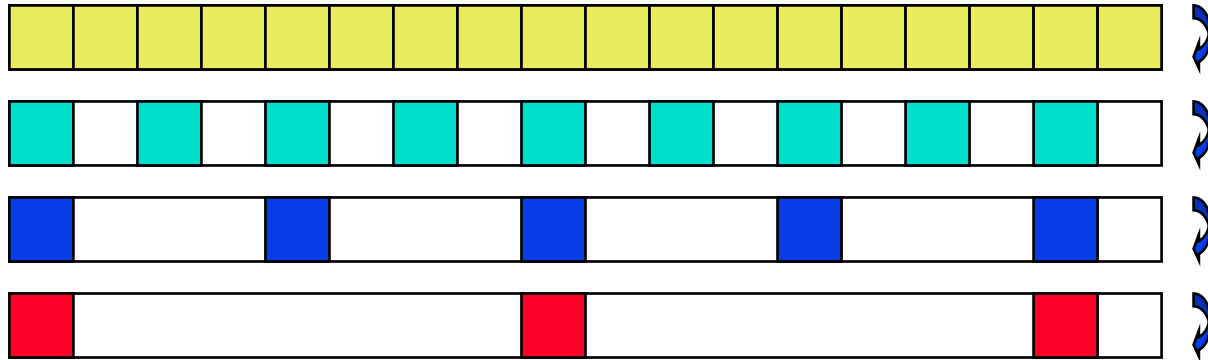
- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
- Consider a tiny cache (for illustration only)

X000	X001
X010	X011
X100	X101
X110	X111

- **Cache line length**: # of bytes loaded together in one entry
- **Associativity**
 - direct-mapped: only one address (line) in a given range in cache
 - *n*-way: 2 or more lines with different addresses exist

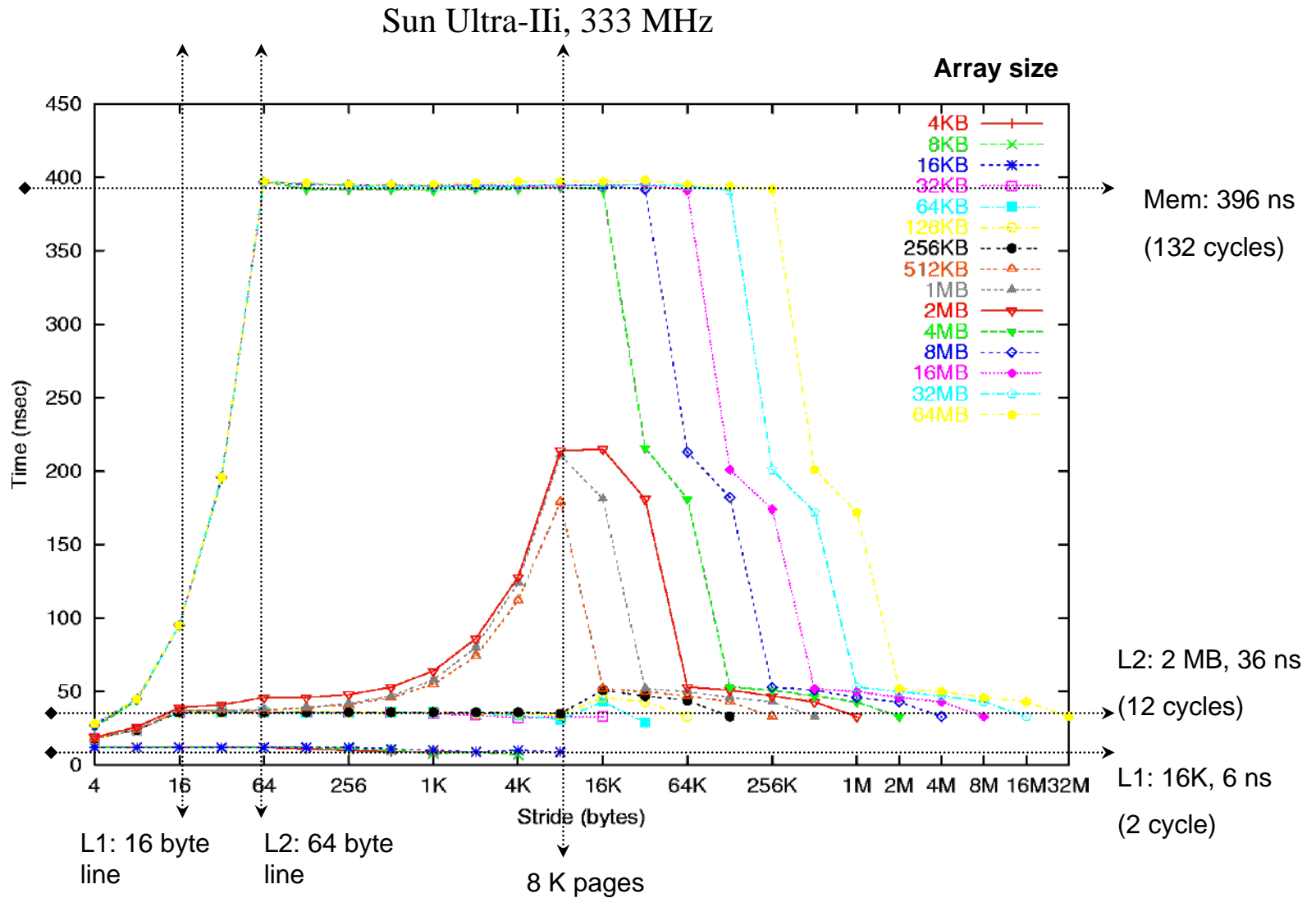
Experimental Study of Memory

- Microbenchmark for memory system performance (Saavedra '92)



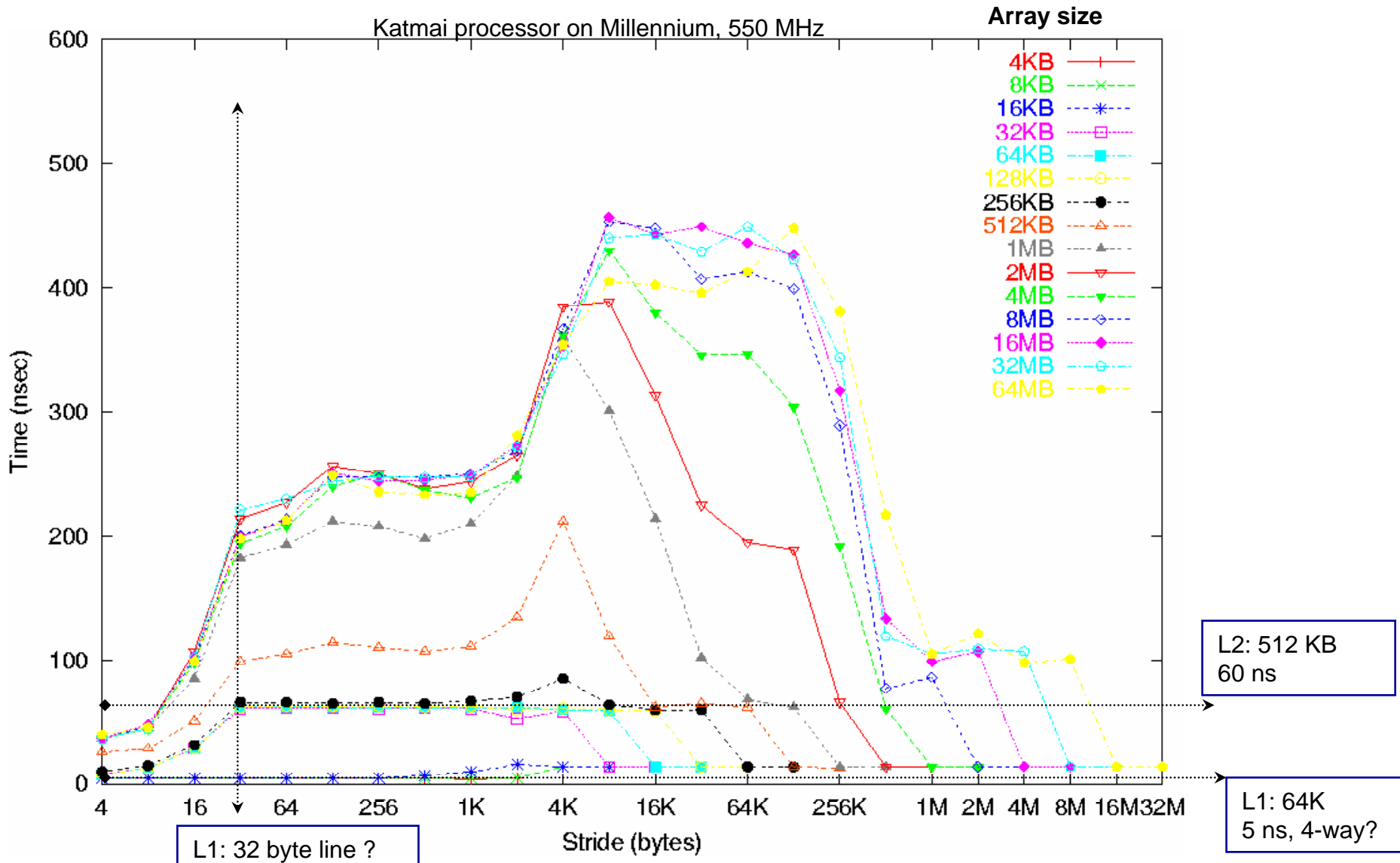
- time the following program for each size(A) and stride s
(repeat to obtain confidence and mitigate timer resolution)
for array A of size from 4KB to 8MB by 2x
for stride s from 8 Bytes (1 word) to size(A)/2 by 2x
for i from 0 to size by s
load A[i] from memory (8 Bytes)

Memory Hierarchy on a Sun Ultra-Ii



See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details
 7/10/2003 CS267 Lecure 2 13

Memory Hierarchy on a Pentium III



Lessons

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

- Next: Example of improving cache performance: **blocking** or **tiling**
 - Idea: decompose problem workload into cache-sized pieces

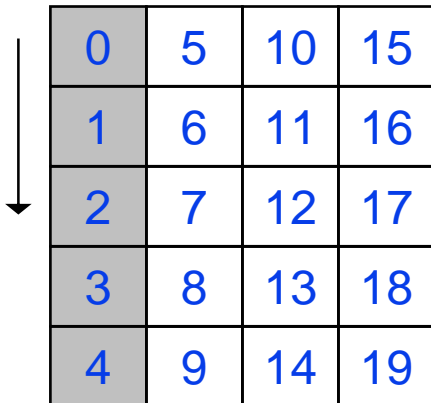
Outline

- Parallelism in Modern Processors
- Memory Hierarchies
- Matrix Multiply Cache Optimizations
- Bag of Tricks

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)
 - by row, or “row major” (C default)

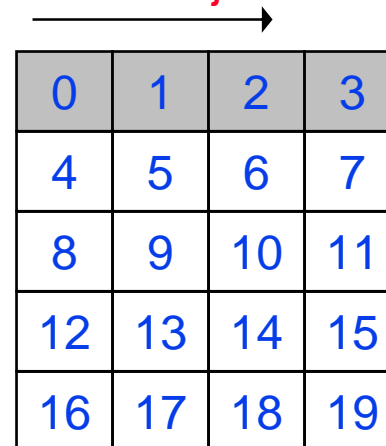
Column major



A 5x4 grid representing column-major storage. The first column (0, 1, 2, 3, 4) is shaded gray. A vertical arrow on the left points downwards, indicating the order of memory addresses.

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

Row major



A 6x4 grid representing row-major storage. The first row (0, 1, 2, 3) is shaded gray. A horizontal arrow above the grid points to the right, indicating the order of memory addresses.

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

Note on “Performance”

- For linear algebra, measure **performance** as rate of execution:
 - Millions of floating point operations per second (**Mflop/s**)
 - Higher is better
 - Comparing Mflop/s is not the same as comparing time *unless* flops are constant!

- **Speedup** taken wrt time
 - Speedup of A over B = (Running time of B) / (Running time of A)

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 element access

Key to algorithm efficiency

- Minimum possible time = $f \cdot t_f$ when all data in fast memory

- Actual time $f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot \left(1 + \frac{t_m}{t_f} \cdot \frac{1}{q} \right)$

- Larger q means time closer to minimum $f \cdot t_f$

Key to machine efficiency

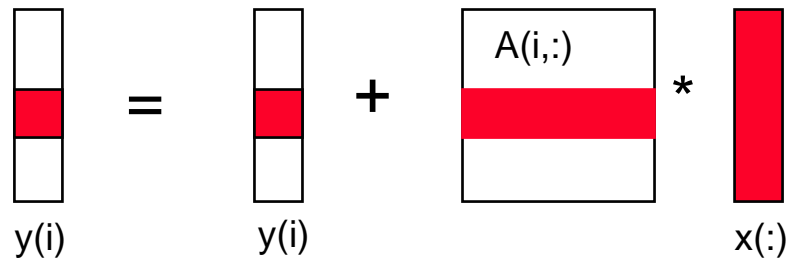
Warm up: Matrix-vector multiplication

```
{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 = $2n^2$
 - $q = f / m \sim 2$
-
- Matrix-vector multiplication limited by slow memory speed

“Naïve” Matrix Multiply

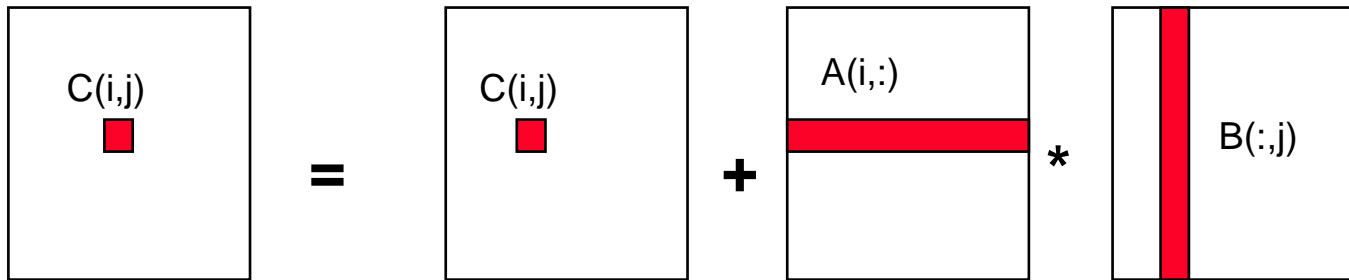
```
{implements C = C + A*B}
```

```
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)
```



“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

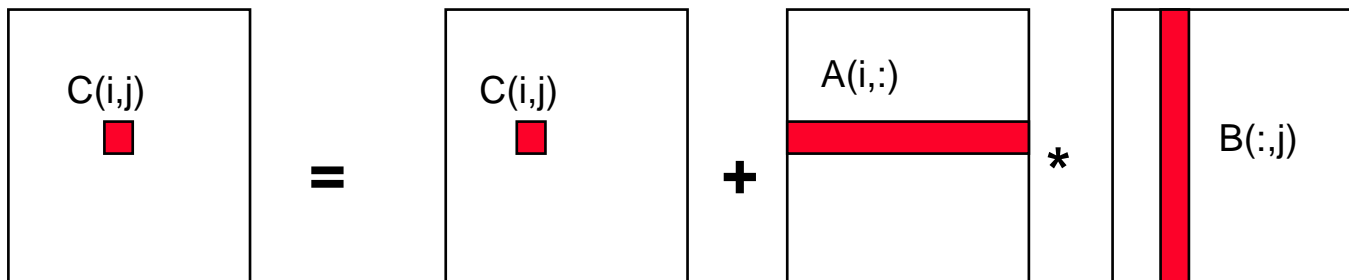
{read $C(i,j)$ into fast memory}

{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

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

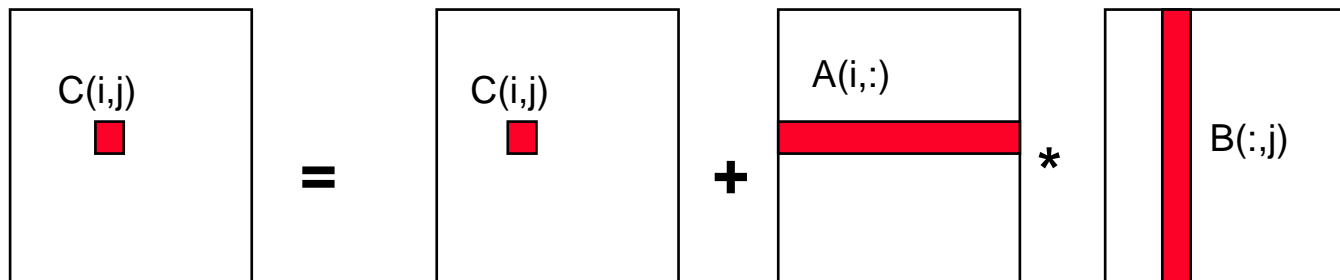
+ n^2 read each row of A once

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

= $n^3 + 3n^2$

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

~ 2 for large n , no improvement over matrix-vector multiply



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}

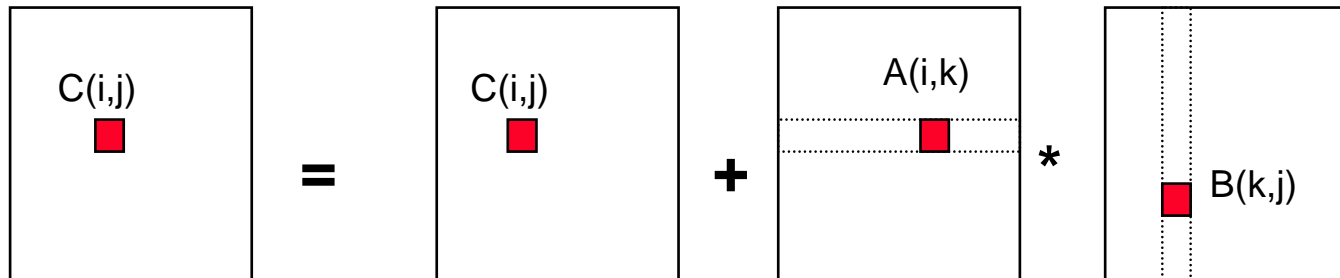
for k = 1 to N

{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 : # of moves from slow to fast memory

Matrix is $n \times n$, and $N \times N$ blocks each of size $b \times b$

$f = 2n^3$ for this problem

$q = f/m$ is algorithmic memory efficiency

So:

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

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

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

So we can improve performance by increasing the block size b

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

Limits to Optimizing Matrix Multiply

Blocked algorithm has ratio $q \sim b$

- Larger block size => faster implementation
- Limit: All three blocks from A, B, C must fit in fast memory (cache):

$$3b^2 \leq M$$

$$\text{So: } q \sim b \leq \text{sqrt}(M/3)$$

Lower bound:

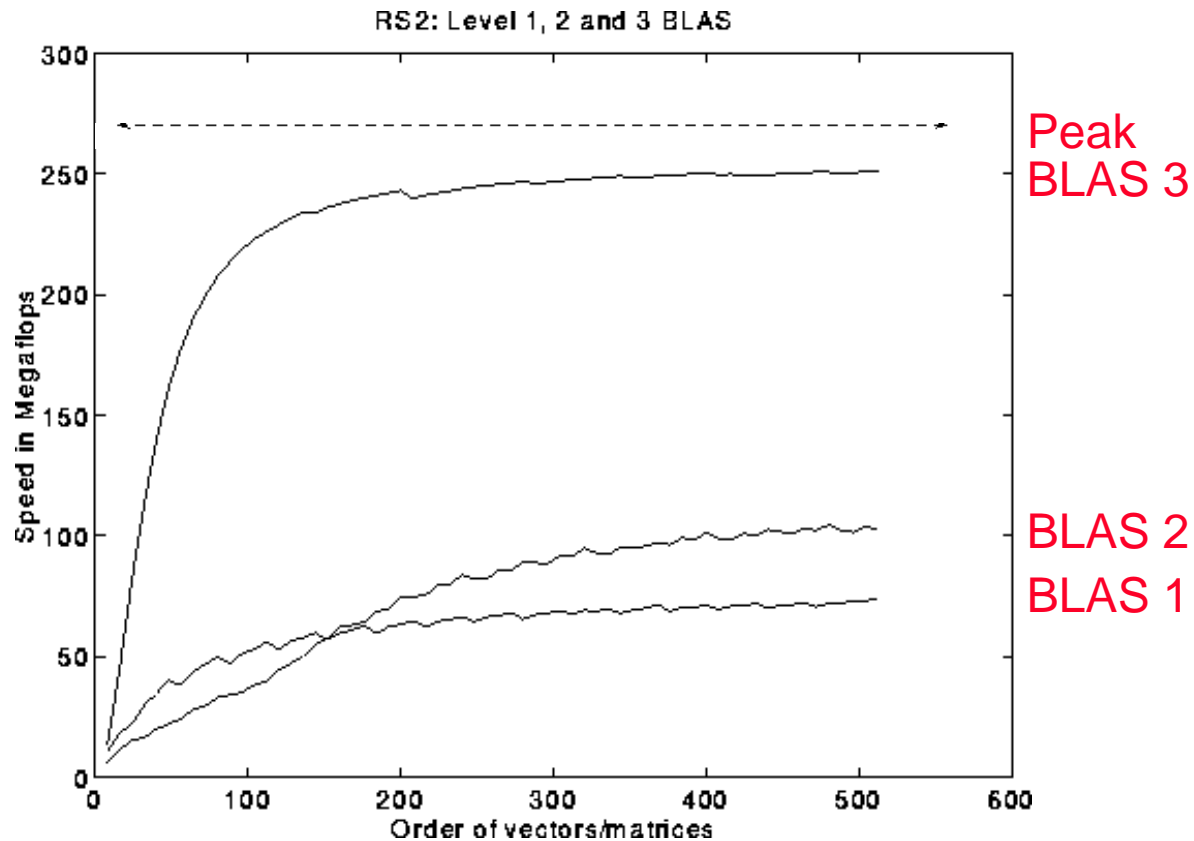
Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (using only algebraic associativity) is limited to: $q = O(\text{sqrt}(M))$

Basic Linear Algebra Subroutines

- Industry standard interface (evolving)
- Hardware vendors, others supply optimized implementations
- History
 - BLAS1 (1970s):
 - vector operations: dot product, saxpy ($y = \alpha * x + y$), etc
 - $m = 2 * n, f = 2 * n, q \sim 1$ 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
 - $m \geq 4n^2, f = O(n^3)$, so q can possibly be as large as n , so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK)
- See www.netlib.org/blas, www.netlib.org/lapack

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)

Locality in Other Algorithms

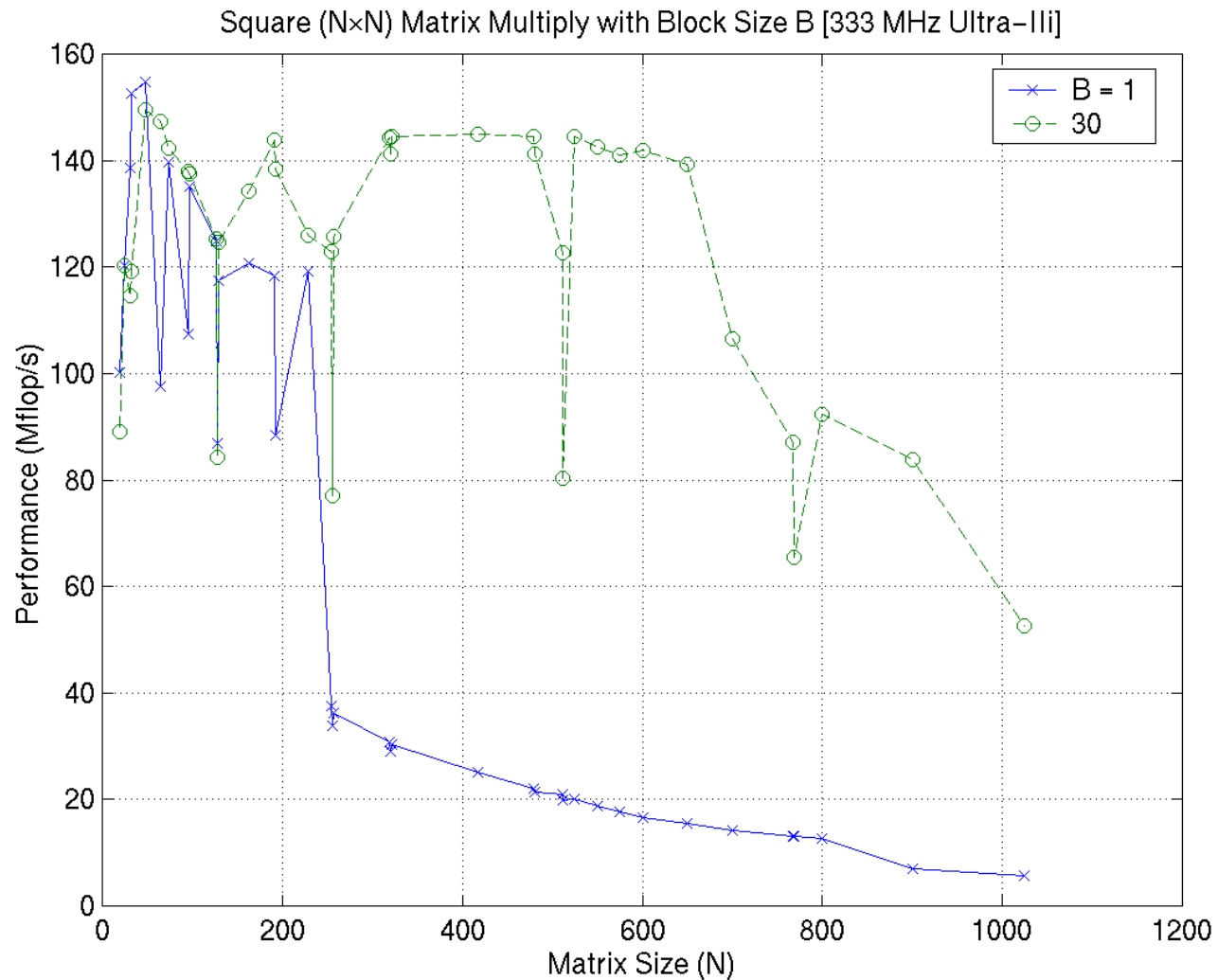
- The performance of any algorithm is limited by q
- In matrix multiply, we increase q by changing computation order
 - increased temporal locality
- For other algorithms and data structures, even hand-transformations are still an open problem
 - sparse matrices (reordering, blocking)
 - trees (B-Trees are for the disk level of the hierarchy)
 - linked lists (some work done here)

Outline

- Parallelism in Modern Processors
- Memory Hierarchies
- Matrix Multiply Cache Optimizations
- Bag of Tricks

Tiling Alone Might Not Be Enough

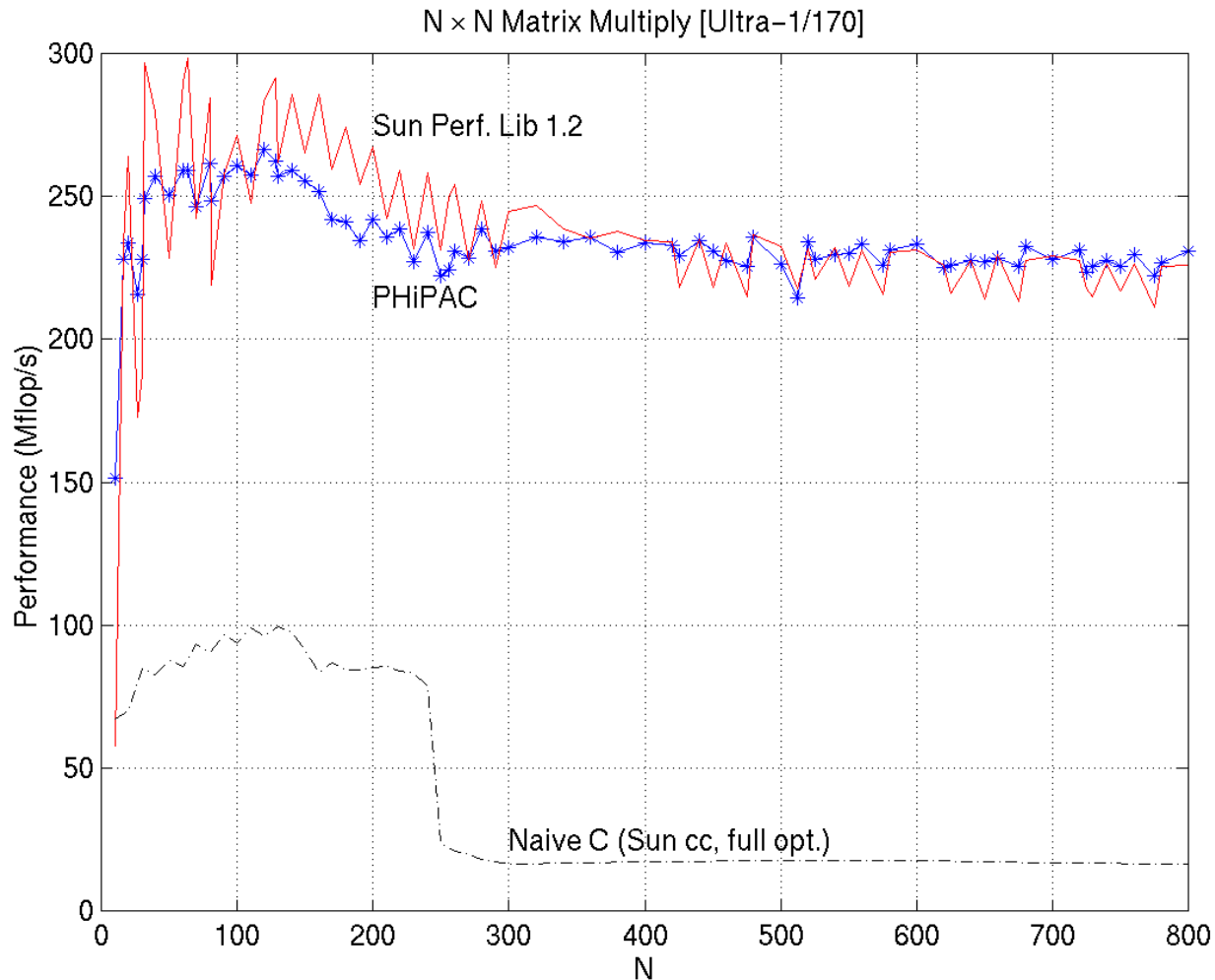
- Naïve and a “naïvely tiled” code



Optimizing in Practice

- Tiling for registers
 - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache
- 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
 - BeBOP: www.cs.berkeley.edu/~richie/bebop
 - PHiPAC: www.icsi.berkeley.edu/~bilmes/hipac
in particular tr-98-035.ps.gz
 - ATLAS: www.netlib.org/atlas

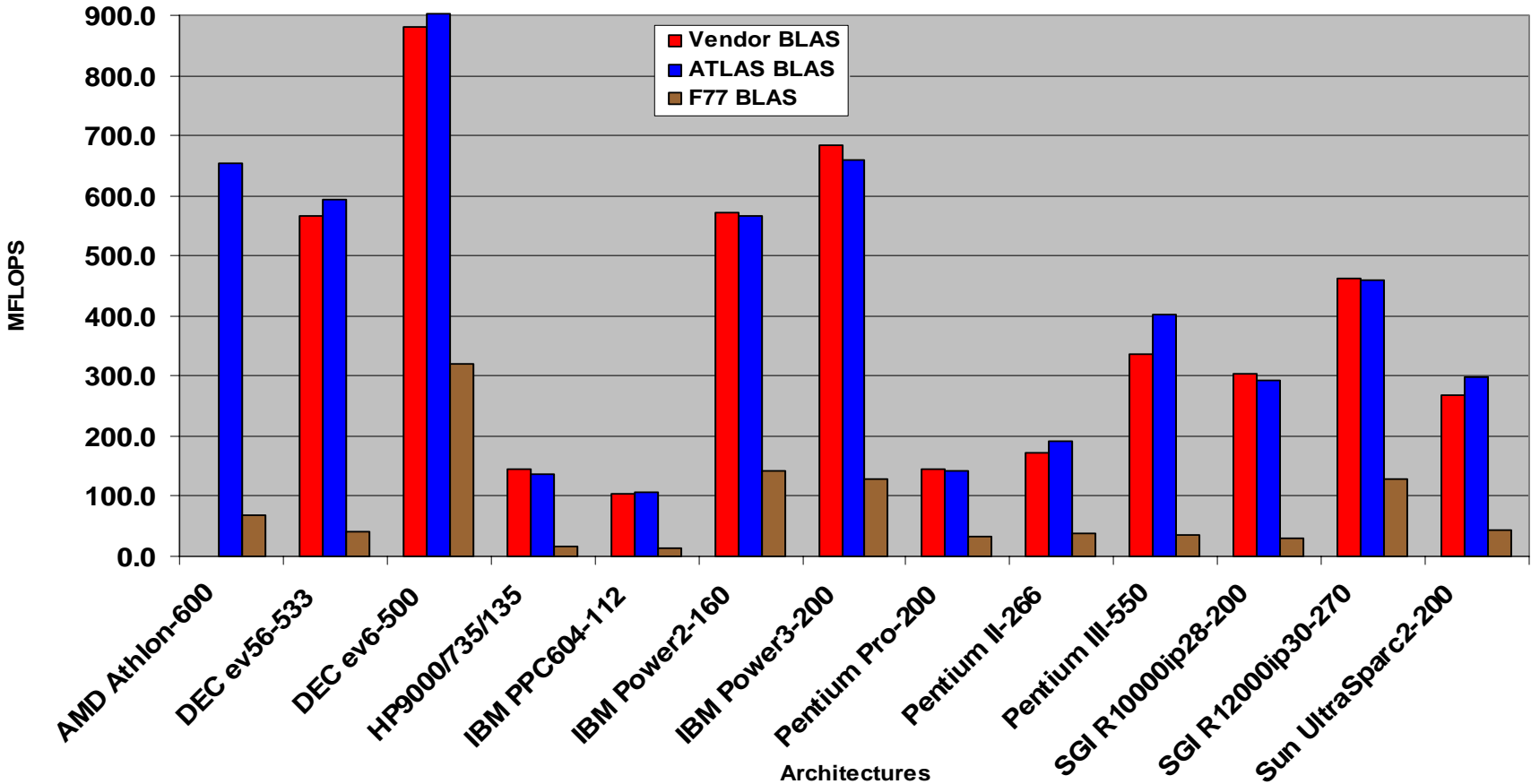
PHiPAC: Portable High Performance ANSI C



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

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.

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 say explicitly (via flag or pragma) that `a` and `b` are not aliased.

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 = ...;

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;
```

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( ... );
```

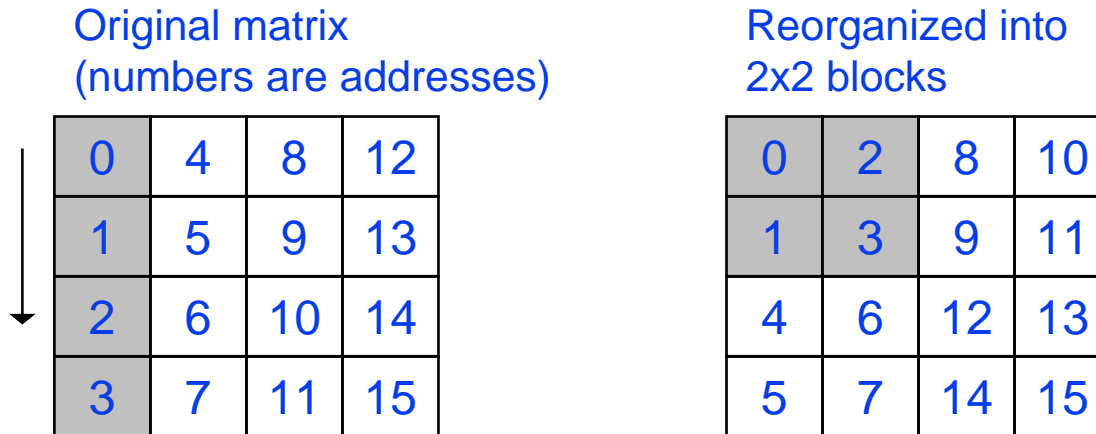
Expose Independent Operations

- 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



Summary

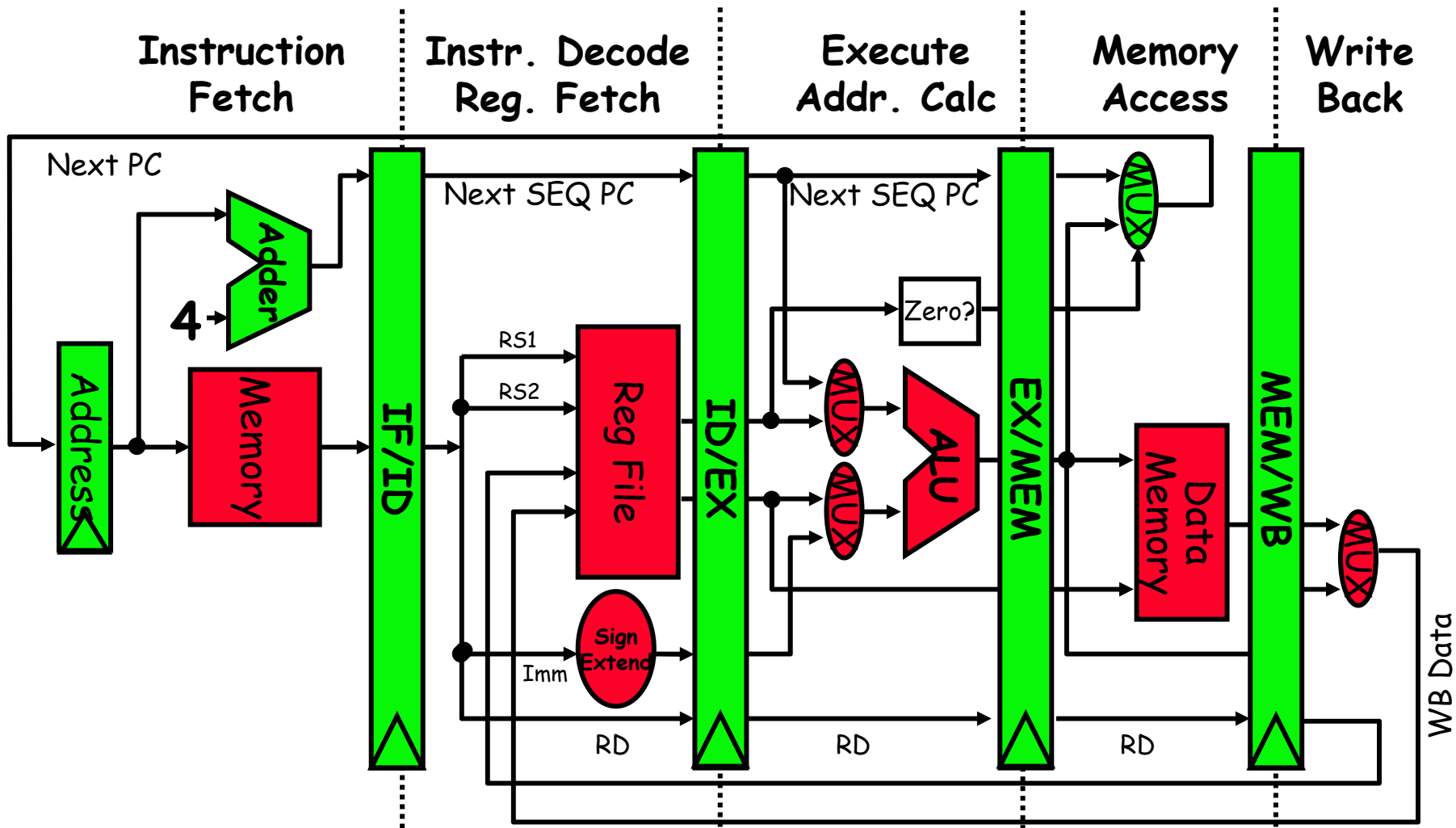
- Performance programming on uniprocessors requires
 - understanding of fine-grained parallelism in processor
 - produce good instruction mix
 - understanding of memory system
 - levels, costs, sizes
 - improve locality
- Blocking (tiling) is a basic approach
 - Techniques apply generally, but the details (e.g., block size) are architecture dependent
 - Similar techniques are possible on other data structures and algorithms
- Now it's your turn: Homework 0 (due 6/25/02)
 - <http://www.cs.berkeley.edu/~richie/bebop/notes/matmul2002>

End

(Extra slides follow)

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

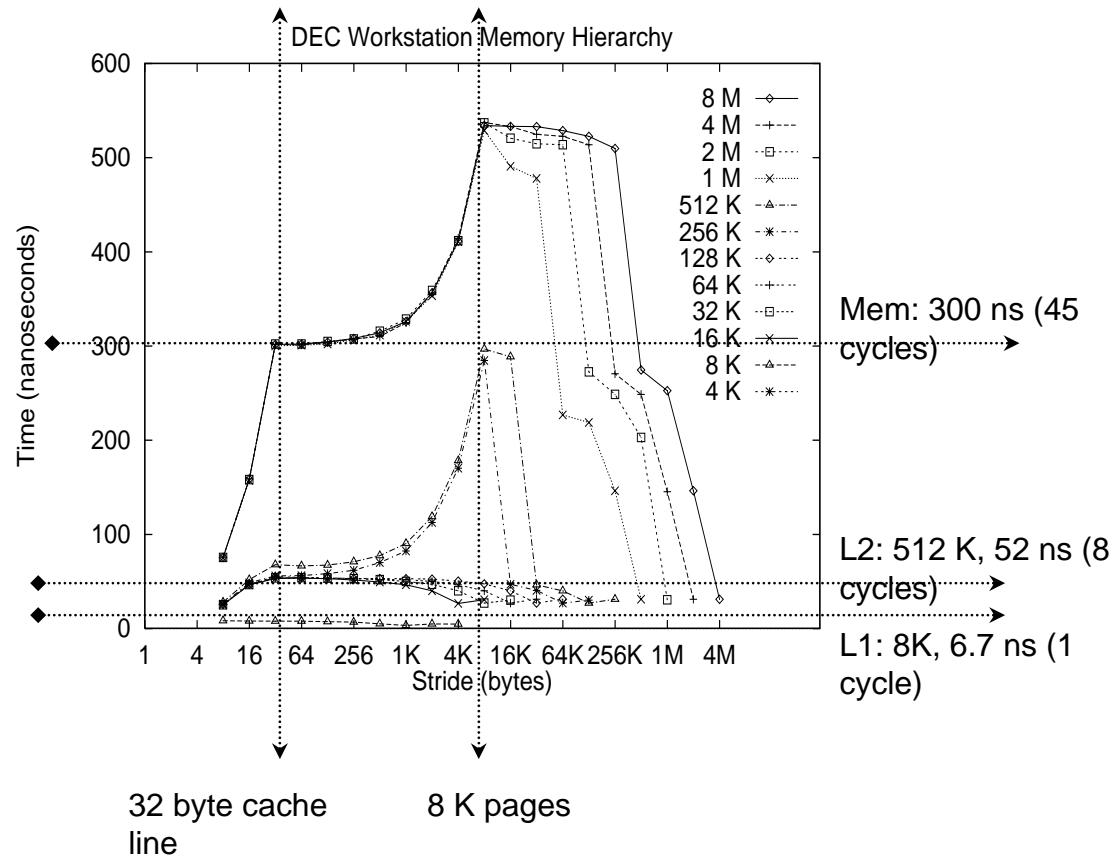
Dependences (Data Hazards) Limit Parallelism

- A **dependence** or **data hazard** is one of the following:
 - **true** of **flow** dependence:
 - a writes a location that b later reads
 - (read-after write or RAW hazard)
 - **anti-dependence**
 - a reads a location that b later writes
 - (write-after-read or WAR hazard)
 - **output** dependence
 - a writes a location that b later writes
 - (write-after-write or WAW hazard)

true	anti	output
a =	= a	a =
= a	a =	a =

Observing a Memory Hierarchy

Dec Alpha, 21064, 150 MHz clock



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