
CS 267 Applications of Parallel Computers

Lecture 2: Memory Hierarchies and Optimizing Matrix Multiplication

Bob Lucas

<http://www.nersc.gov/~dhbailey/cs267>

Millennium Accounts

- To get Millennium accounts, students need to get EECS Instructional Accounts
- Instructions are outlined on the class resource page
- Please mail Jason at ejr@cs.berkeley.edu once you've requested an account and returned the account form.

- Another recommended reference:
 “Industrial Strength Parallel Computing”
 Edited by Alice E. Koniges

Outline

- **Understanding Caches**
- **Optimizing Matrix Multiplication**

Idealized Uniprocessor Model

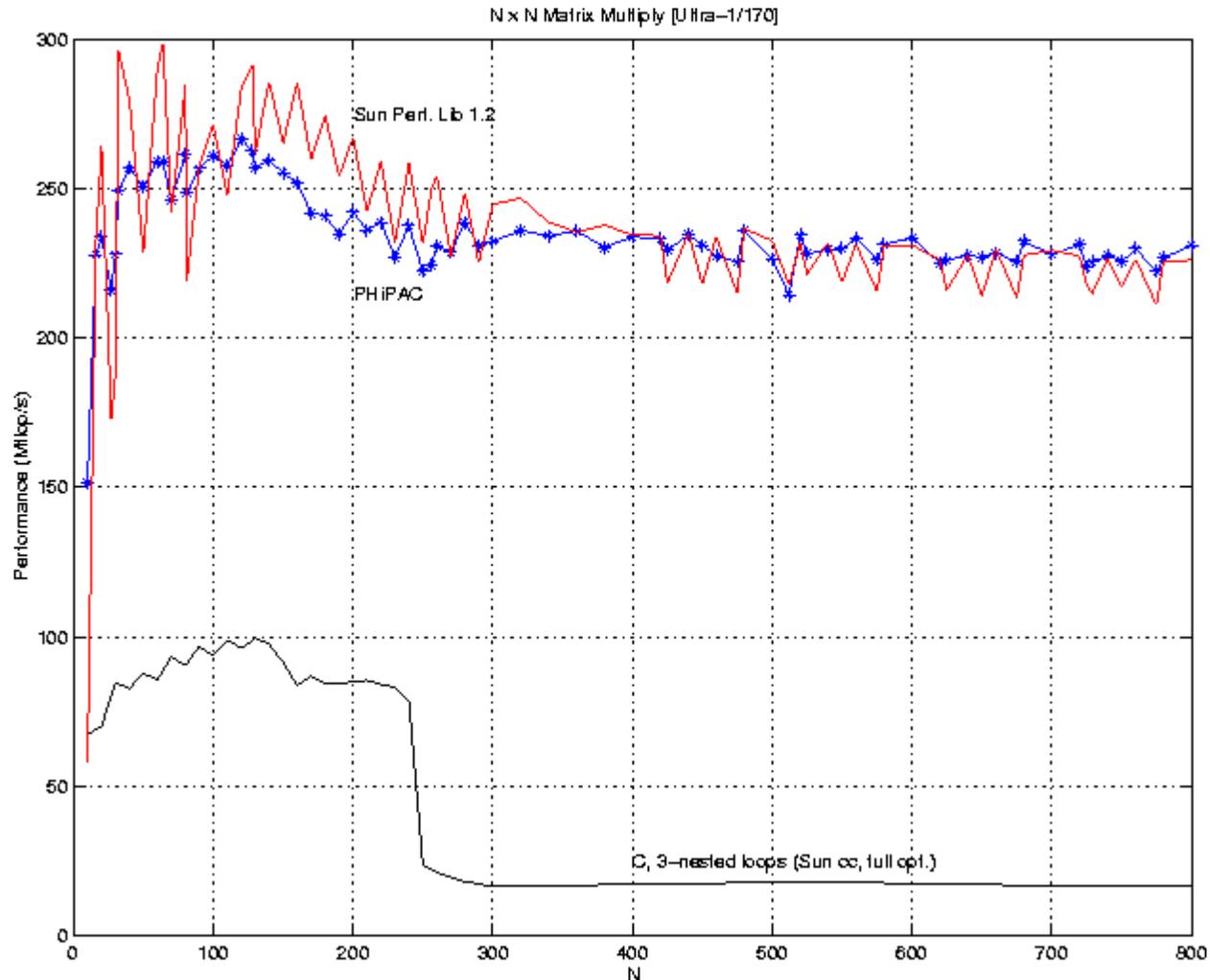
- **Processor can name objects in a simple, flat address space**
 - these represent integers, floats, pointers, structures, arrays, etc.
 - exist in the program stack, static region, or heap
- **Operations include**
 - read and write from memory (given an address/pointer)
 - arithmetic and other logical operations
- **Order specified by program**
 - read returns the most recently written data
 - compiler and architecture may reorder operations to optimize performance, as long as the programmer cannot see any reordering
- **Performance**
 - each operation has roughly the same cost (read, write, multiply, etc.)

Uniprocessor Reality

- **Modern processors use a variety of techniques for performance**
 - **caches**
 - **small amount of fast memory where values are “cached” in hope of reusing recently used or nearby data**
 - **different memory ops can have very different costs**
 - **parallelism**
 - **superscalar processors have 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 understand all of this and can optimize your program; in practice they don't.**

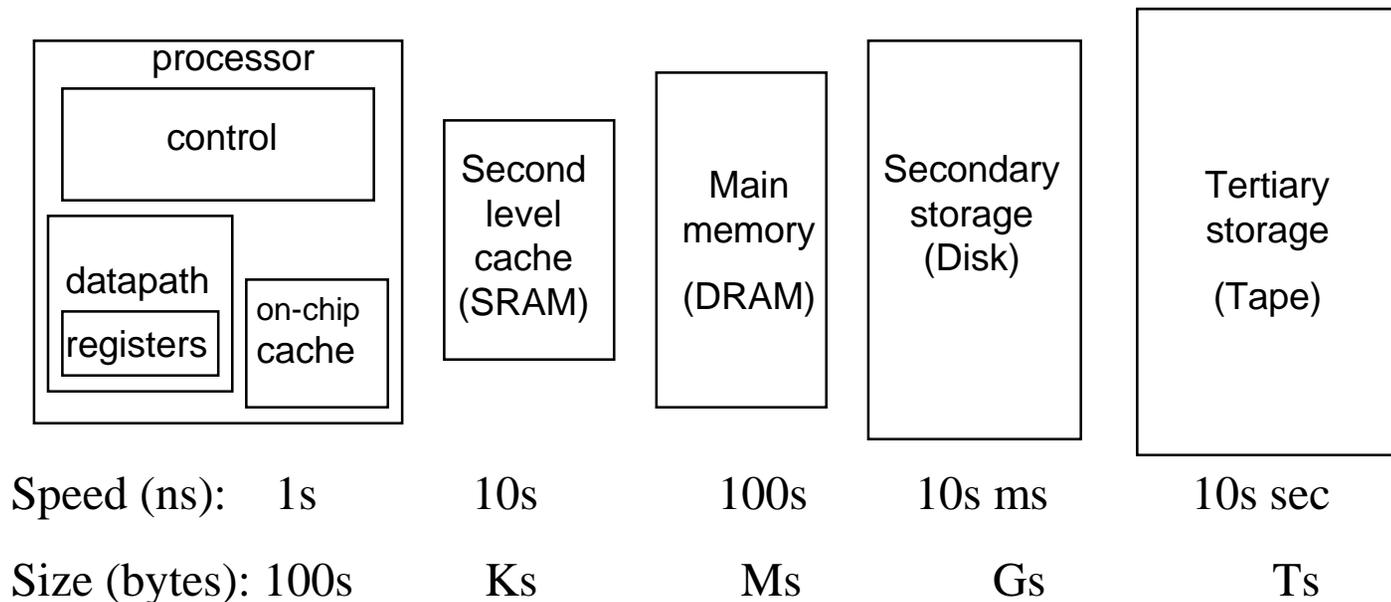
Matrix-multiply, optimized several ways



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

Memory Hierarchy

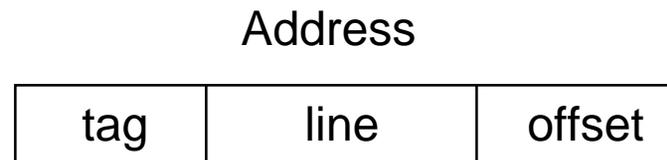
- Most programs have a high degree of **locality** in their accesses
 - spatial locality: accessing things nearby previous accesses
 - temporal locality: reusing an item that was previously accessed
- Memory Hierarchy tries to exploit locality



Cache Basics

- **Cache hit:** a memory access that is found in the cache -- cheap
- **Cache miss:** a memory access that is not in the cache - expensive, because we need to get the data from elsewhere
- **Consider a tiny cache (for illustration only)**

X000	X001
X010	X011
X100	X101
X110	X111

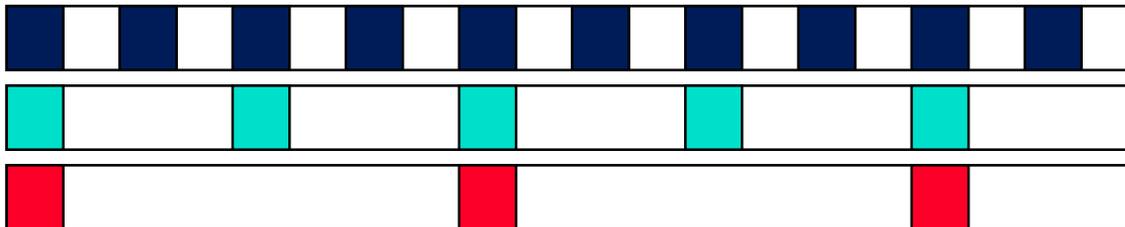


- **Cache line length:** number of bytes loaded together in one entry
- **Direct mapped:** only one address (line) in a given range in cache
- **Associative:** 2 or more lines with different addresses exist

Experimental Study of Memory

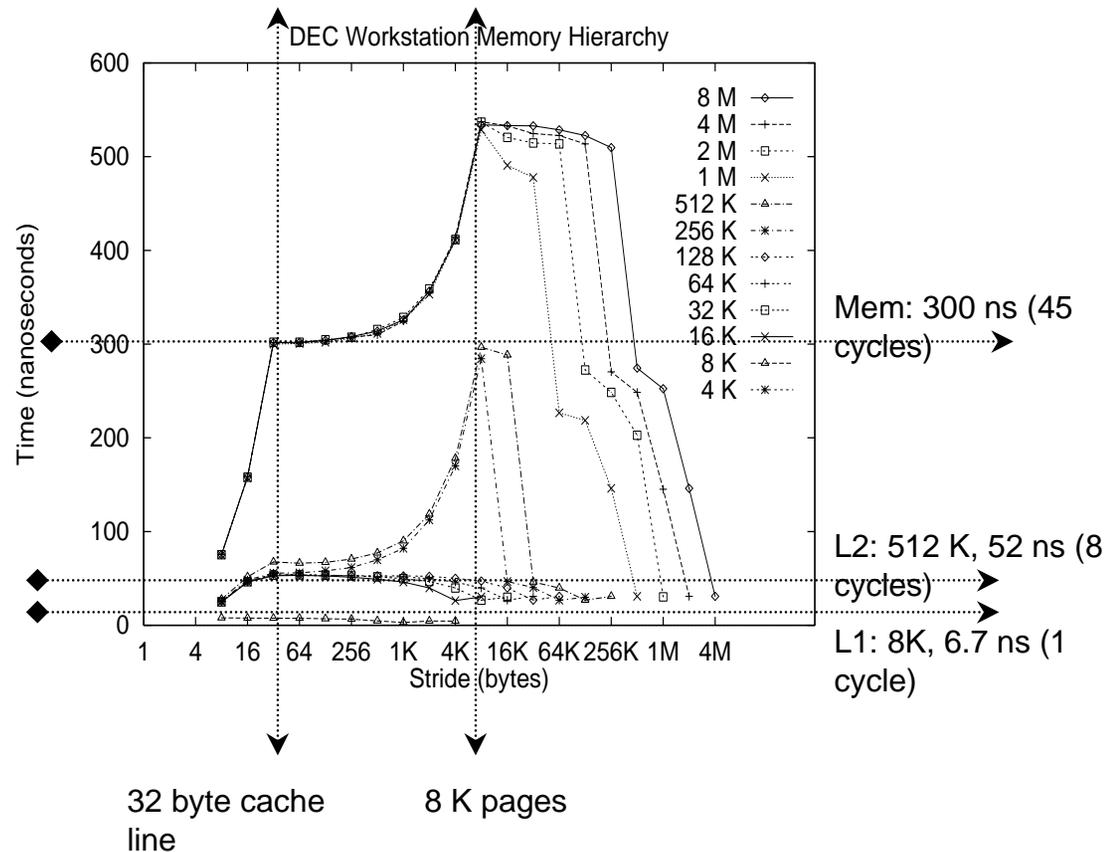
◦ Microbenchmark for memory system performance

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)



Observing a Memory Hierarchy

Dec Alpha, 21064, 150 MHz clock



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

Lessons

- The 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
- Since we want to write fast programs, we must take the architecture into account, even on uniprocessors
- Since the actual performance is so complicated, we need simple models to help us design efficient algorithms
- We will illustrate with a common technique for improving cache performance, called **blocking**

Optimizing Matrix Addition for Caches

- Dimension $A(n,n)$, $B(n,n)$, $C(n,n)$
- A , B , C stored by column (as in Fortran)
- Algorithm 1:
 - for $i=1:n$, for $j=1:n$, $A(i,j) = B(i,j) + C(i,j)$
- Algorithm 2:
 - for $j=1:n$, for $i=1:n$, $A(i,j) = B(i,j) + C(i,j)$
- What is “memory access pattern” for Algs 1 and 2?
- Which is faster?
- What if A , B , C stored by row (as in C)?

Using a Simpler 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 element access
- Minimum possible Time = $f \cdot t_f$, when all data in fast memory
- Actual Time = $f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot (1 + (t_m/t_f) \cdot (1/q))$
- Larger q means Time closer to minimum $f \cdot t_f$

Simple example using memory model

- To see results of changing q , consider simple computation

$$s = 0$$

for $i = 1, n$

$$s = s + h(X[i])$$

- Assume $t_f=1$ Mflop/s on fast memory
- Assume moving data is $t_m = 10$
- Assume h takes q flops
- Assume array X is in slow memory

- So $m = n$ and $f = q*n$
- Time = read X + compute = $10*n + q*n$
- Mflop/s = $f/t = q/(10 + q)$
- As q increases, this approaches the “peak” speed of 1 Mflop/s

Simple Example (continued)

◦ Algorithm 1

$s1 = 0; s2 = 0$

for $j = 1$ to n

$s1 = s1 + h1(X(j))$

$s2 = s2 + h2(X(j))$

◦ Algorithm 2

$s1 = 0; s2 = 0$

for $j = 1$ to n

$s1 = s1 + h1(X(j))$

for $j = 1$ to n

$s2 = s2 + h2(X(j))$

◦ Which is faster?

Optimizing Matrix Multiply for Caches

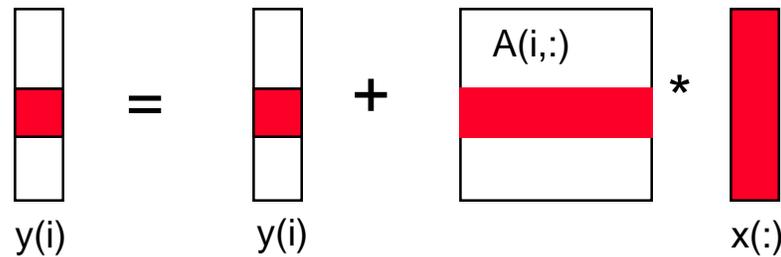
- **Several techniques for making this faster on modern processors**
 - heavily studied
- **Some optimizations done automatically by compiler, but can do much better**
- **In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations**
 - BLAS = Basic Linear Algebra Subroutines
- **Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques**

Warm up: Matrix-vector multiplication $y = y + A*x$

for $i = 1:n$

for $j = 1:n$

$$y(i) = y(i) + A(i,j)*x(j)$$



Warm up: Matrix-vector multiplication $y = y + A*x$

{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 = $3*n + n^2$
- f = number of arithmetic operations = $2*n^2$
- $q = f/m \approx 2$
- Matrix-vector multiplication limited by slow memory speed

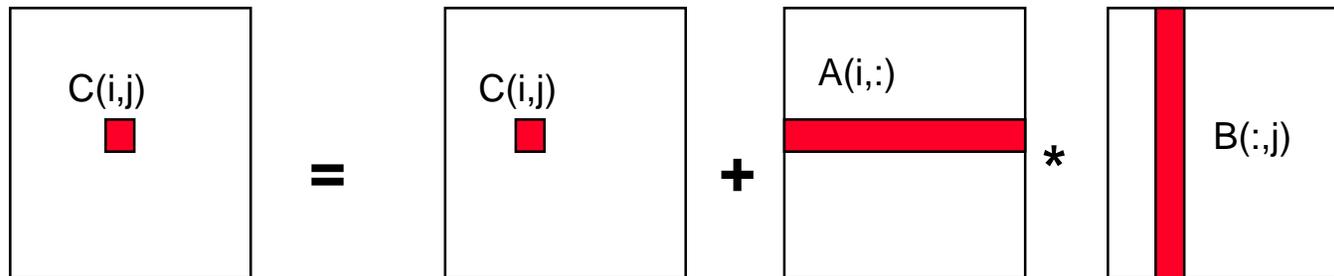
Matrix Multiply $C=C+A*B$

for $i = 1$ to n

for $j = 1$ to n

for $k = 1$ to n

$$C(i,j) = C(i,j) + A(i,k) * B(k,j)$$



Matrix Multiply $C=C+A*B$ (unblocked, or untiled)

for $i = 1$ to n

{read row i of A into fast memory}

for $j = 1$ to n

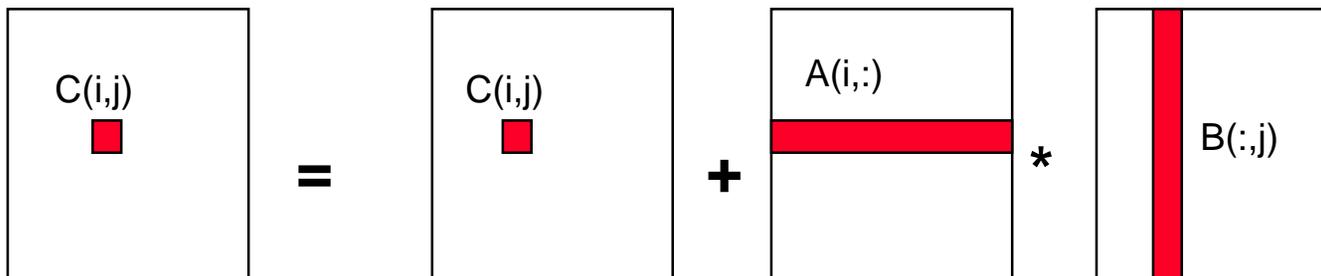
{read $C(i,j)$ into fast memory}

{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}



Matrix Multiply (unblocked, or untiled)

Number of slow memory references on unblocked matrix multiply

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

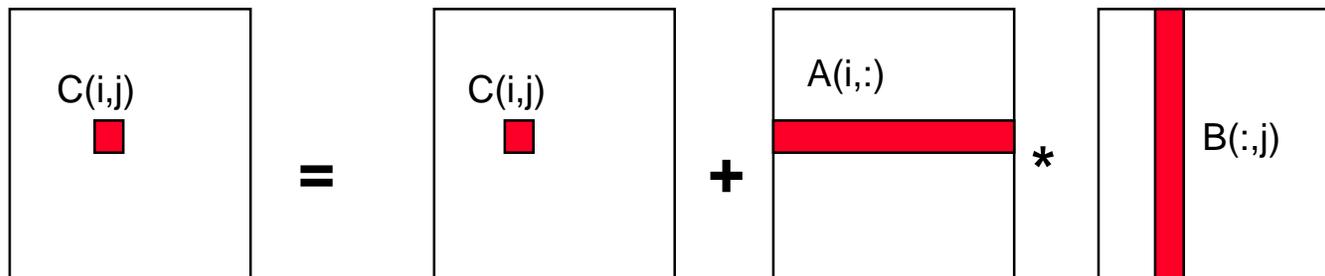
+ n^2 read each column of A once for each i

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

= $n^3 + 3 \cdot n^2$

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

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



Matrix Multiply (blocked, or tiled)

Consider A,B,C to be N by N matrices of b by b subblocks where $b=n/N$ is called the **blocksize**

for i = 1 to N

for j = 1 to N

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

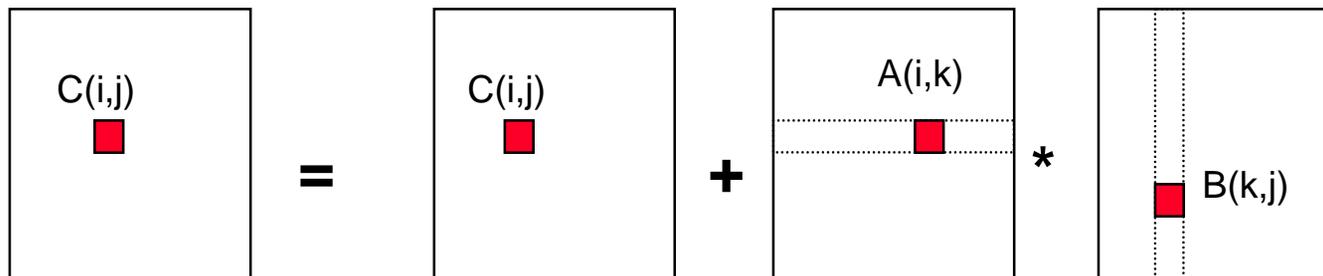
for k = 1 to N

{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}



Matrix Multiply (blocked or tiled)

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply

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

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

$$\sim n/N = b \quad \text{for large } n$$

So we can improve performance by increasing the blocksize b

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

Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large: $3 \cdot b^2 \leq M$, so $q \sim b \leq \sqrt{M/3}$

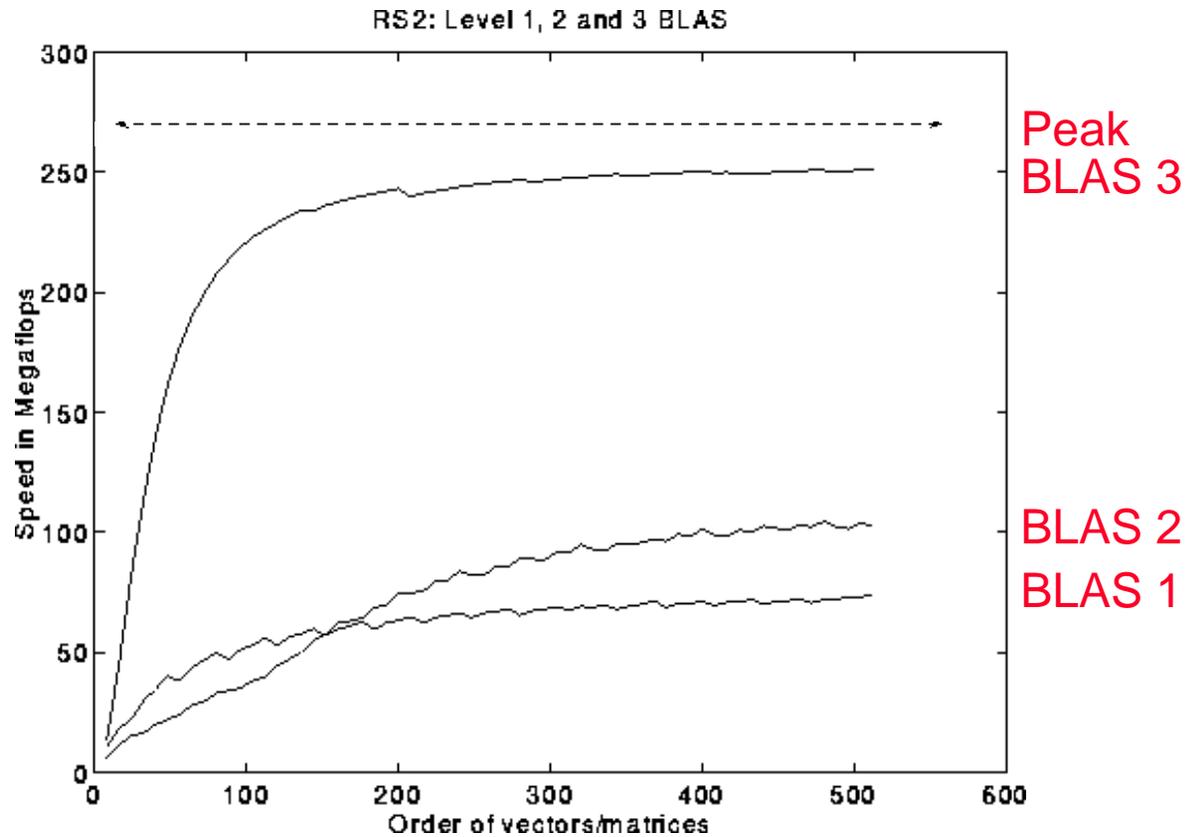
Theorem (Hong, Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O(\sqrt{M})$

More on BLAS (Basic Linear Algebra Subroutines)

- Industry standard interface (evolving)
- 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)
- www.netlib.org/blas, www.netlib.org/lapack

BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

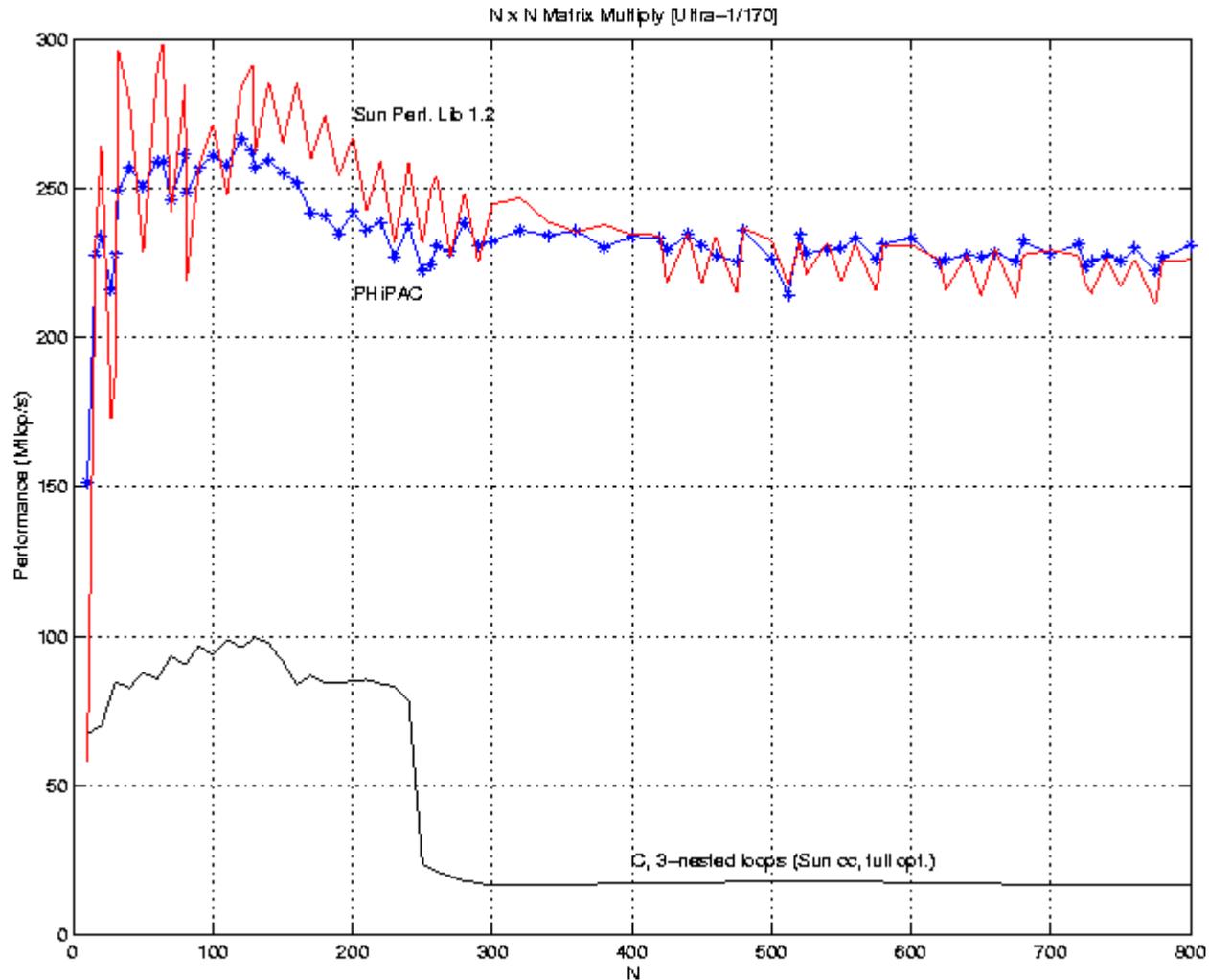


BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

Optimizing in practice

- **Tiling for registers**
 - loop unrolling, use of named “register” variables
- **Tiling for multiple levels of cache**
- **Exploiting fine-grained parallelism within the processor**
 - super scalar
 - pipelining
- **Complicated compiler interactions**
- **Hard to do by hand (but you’ll try)**
- **Automatic optimization an active research area**
 - PHIPAC: www.icsi.berkeley.edu/~bilmes/hipac
 - www.cs.berkeley.edu/~iyer/ascii_slides.ps
 - ATLAS: www.netlib.org/atlas/index.html

PHIPAC: Portable High Performance ANSI C



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

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 8 multiplies

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

$$\text{Let } p_1 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

$$p_2 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$p_3 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

$$p_4 = (a_{11} + a_{12}) * b_{22}$$

$$p_5 = a_{11} * (b_{12} - b_{22})$$

$$p_6 = a_{22} * (b_{21} - b_{11})$$

$$p_7 = (a_{21} + a_{22}) * b_{11}$$

$$\text{Then } m_{11} = p_1 + p_2 - p_4 + p_6$$

$$m_{12} = p_4 + p_5$$

$$m_{21} = p_6 + p_7$$

$$m_{22} = p_2 - p_3 + p_5 - p_7$$

Extends to nxn by divide&conquer

Strassen (continued)

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

- Why does Hong/Kung theorem not apply?
- Available in several libraries
- Up to several times faster if n large enough (100s)
- Needs more memory than standard algorithm
- Can be less accurate because of roundoff error
- Current world's record is $O(n^{2.376..})$

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)

Summary

- **Performance programming on uniprocessors requires**
 - understanding of memory system
 - levels, costs, sizes
 - understanding of fine-grained parallelism in processor to produce good instruction mix
- **Blocking (tiling) is a basic approach that can be applied to many matrix algorithms**
- **Applies to uniprocessors and parallel processors**
 - The technique works for any architecture, but choosing the blocksize b and other details depends on the architecture
- **Similar techniques are possible on other data structures**
- **You will get to try this in Assignment 2 (see the class homepage)**