

---

# CS 267: Optimizing for Uniprocessors—A Case Study in Matrix Multiplication

Katherine Yelick

yelick@cs.berkeley.edu

<http://www.cs.berkeley.edu/~yelick/cs267>

# Recap and Plan

---

- Scaled speedup: operate near the memory boundary
- Memory systems on modern processors are complicated.
- The performance of a simple program can depend on the details of the micro-architecture.
- Today we will study matrix multiplication optimizations
  - An important kernel in some scientific problems
  - 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 well-studied algorithm in high performance computing

# Outline

---

- Performance Modeling
- Matrix-Vector Multiply (Warmup)
- Matrix Multiply Cache Optimizations
- Bag of Tricks
- Research in Matrix Multiply

# 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
- Minimum possible time =  $f * t_f$  when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 + \frac{t_m}{t_f} * 1/q)$
- Larger  $q$  means time closer to minimum  $f * t_f$

**Computational Intensity: Key to algorithm efficiency**

**Key to machine efficiency**

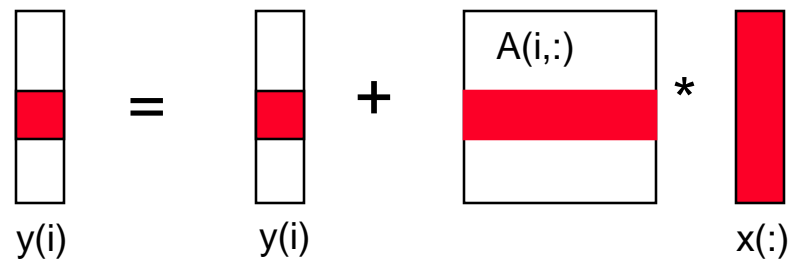
# Warm up: Matrix-vector multiplication

```
{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
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$
  - $= 2 * n^2 * (1 + 0.5 * t_m/t_f)$
- For  $t_f$  and  $t_m$ , using data from R. Vuduc's PhD (pp 352-3)
  - <http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf>
  - For  $t_m$  use words-per-cache-line / minimum-memory-latency

	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  
peak  
speed)

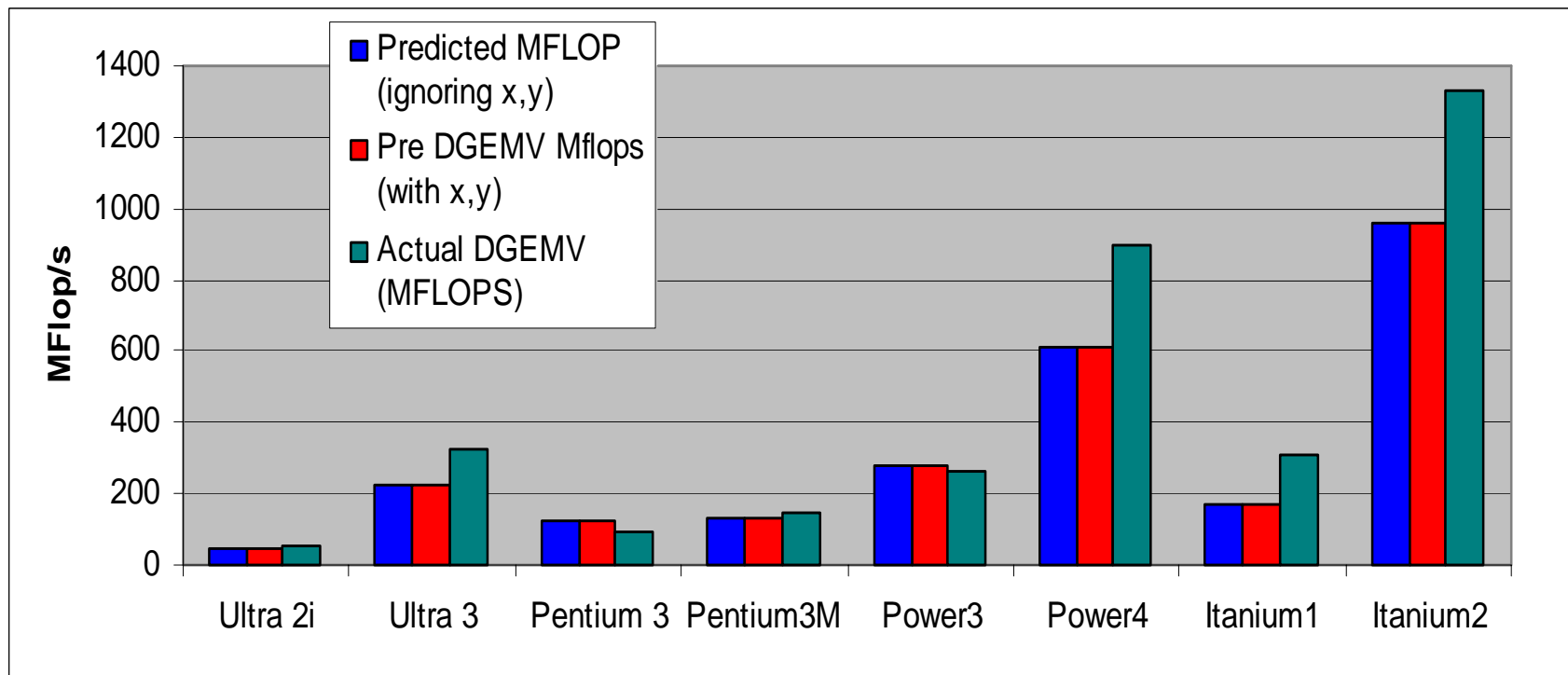
# What Simplifying Assumptions

---

- 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?
  - Using 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}
```

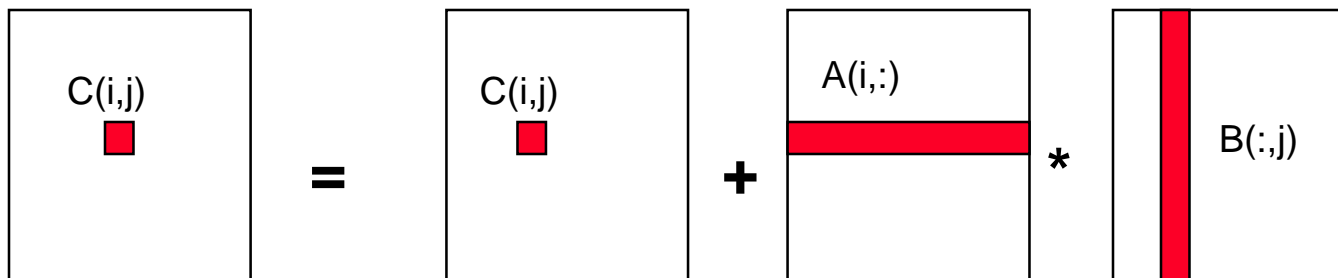
```
for i = 1 to n
```

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

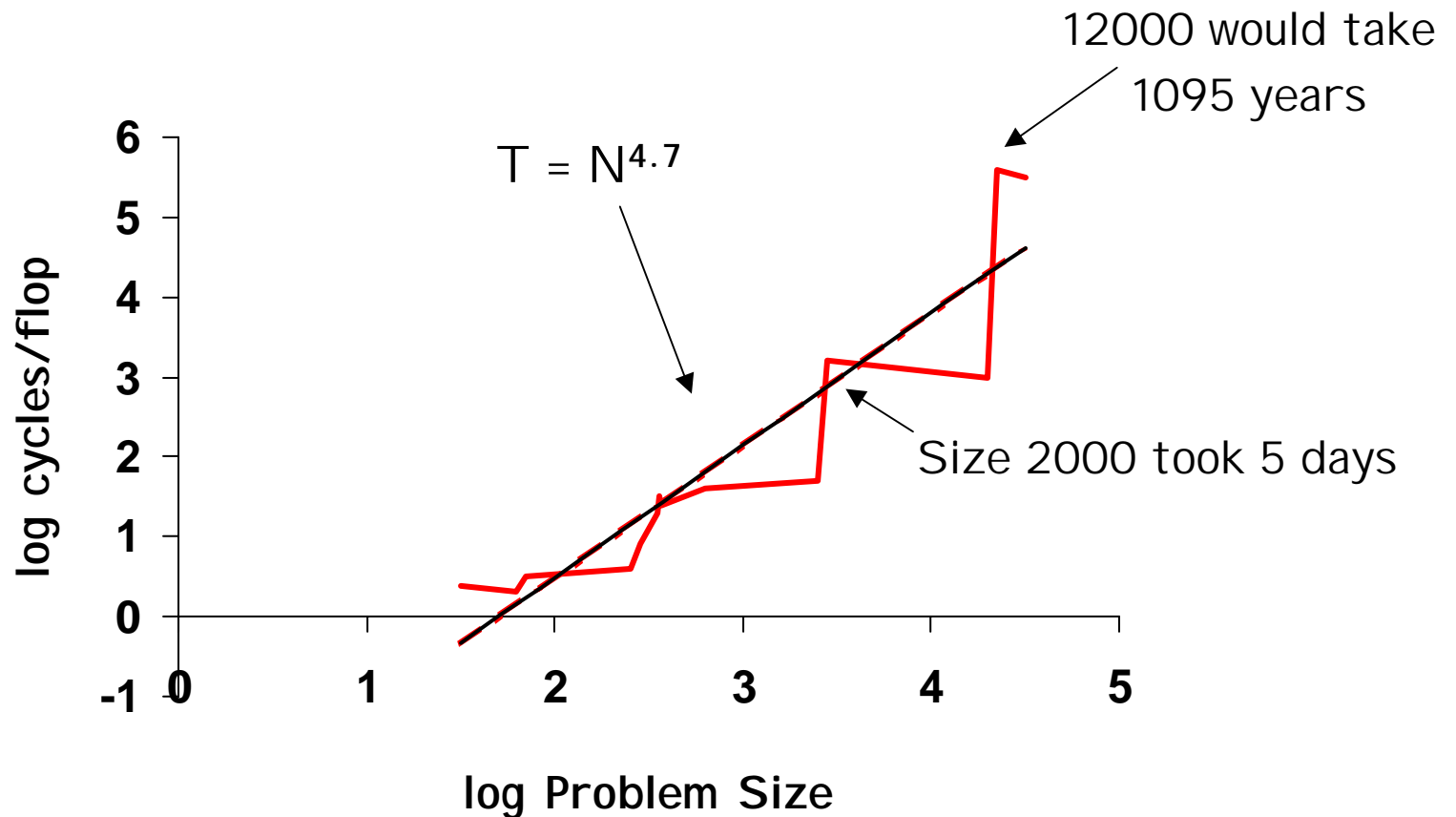
```
    for k = 1 to n
```

```
      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



# Matrix Multiply on RS/6000



$O(N^3)$  performance would have constant cycles/flop  
Performance looks much closer to  $O(N^5)$

# 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  $A+i*n+j$

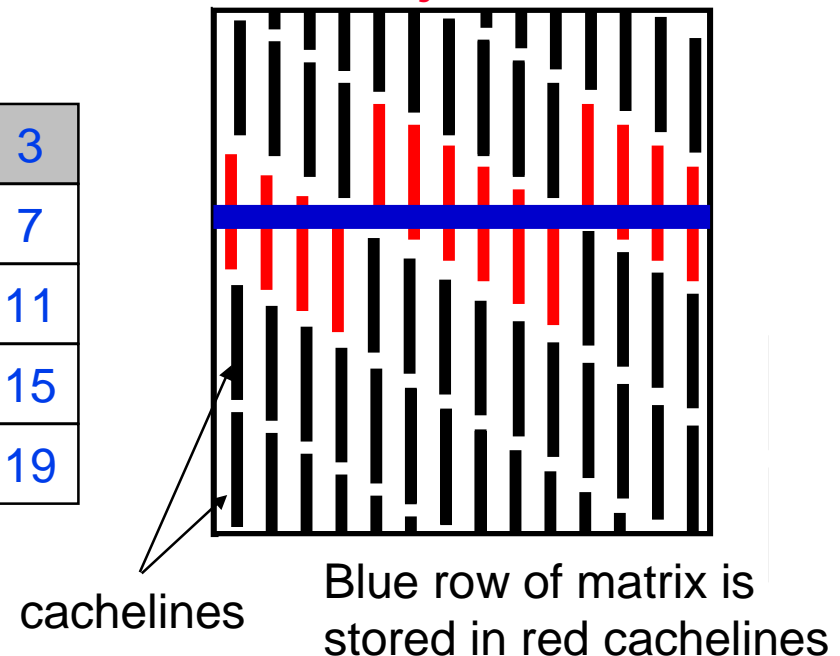
**Column major**

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**



# “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)
```

Reuse  
value from a  
register

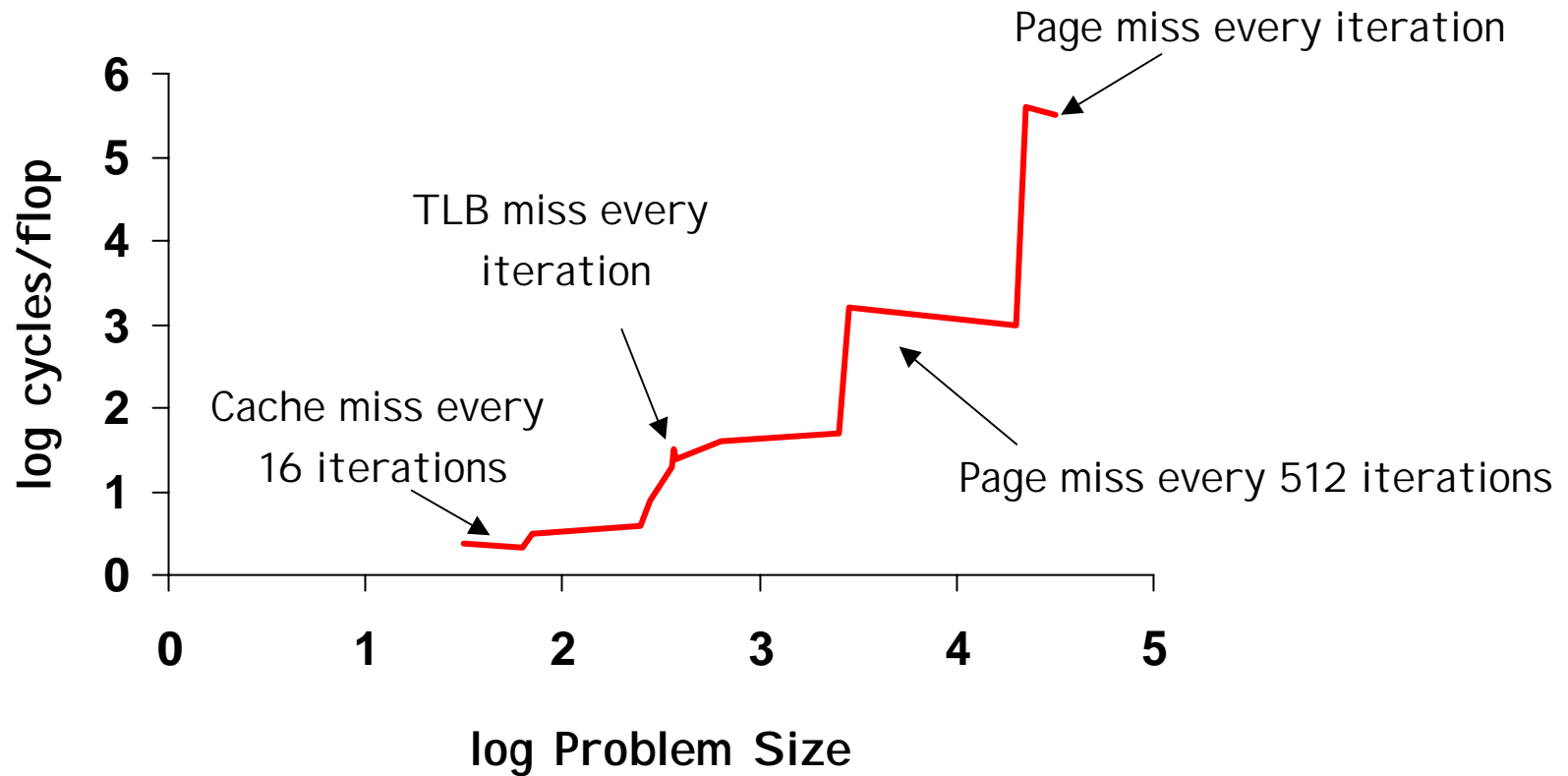
Stride-N  
access to  
one row\*

Sequential  
access through  
entire matrix

- **When cache (or TLB or memory) can't hold entire B matrix, there will be a miss on every line.**
- **When cache (or TLB or memory) can't hold a row of A, there will be a miss on each access**

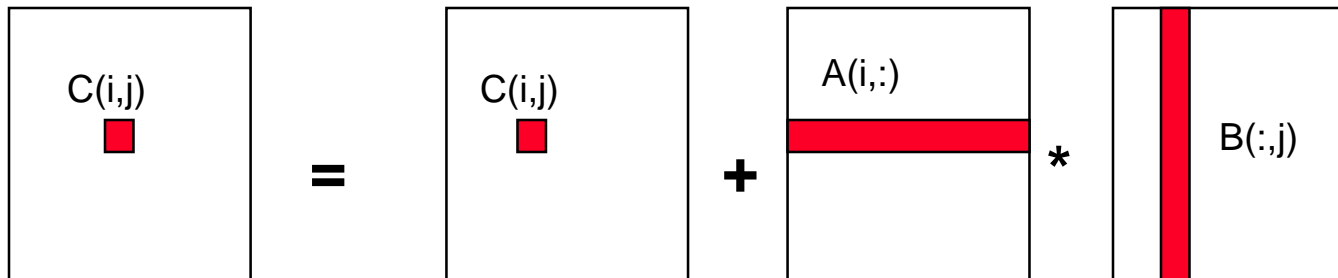
\*Assumes column-major order

# Matrix Multiply on RS/6000



# “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

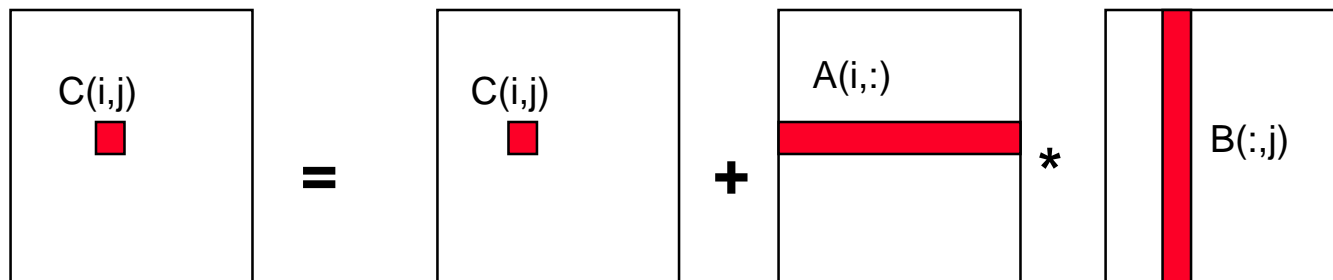
+  $n^2$  read each row of A once

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

=  $n^3 + 3n^2$

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

$\sim 2$  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}

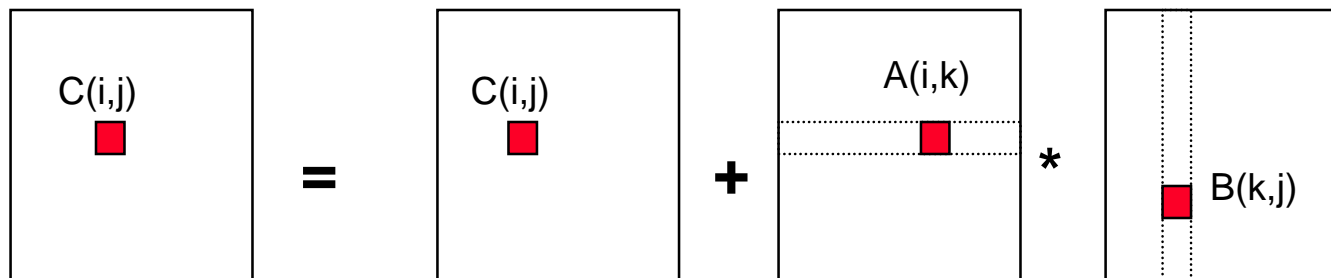
for k = 1 to N

{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:

$m$  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:

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

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

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

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  $M_{fast}$

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

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

$$M_{fast} \geq 3b^2 \approx 3q^2 = 3(T_m/T_f)^2$$

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

	$t_m/t_f$	required KB
Ultra 2i	24.8186	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 associativity
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \approx b \leq \sqrt{M_{\text{fast}}/3}$$

- There is a lower bound result that says we cannot do any better than this (using only algebraic associativity)
- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only algebraic associativity) is limited to  $q = O(\sqrt{M_{\text{fast}}})$

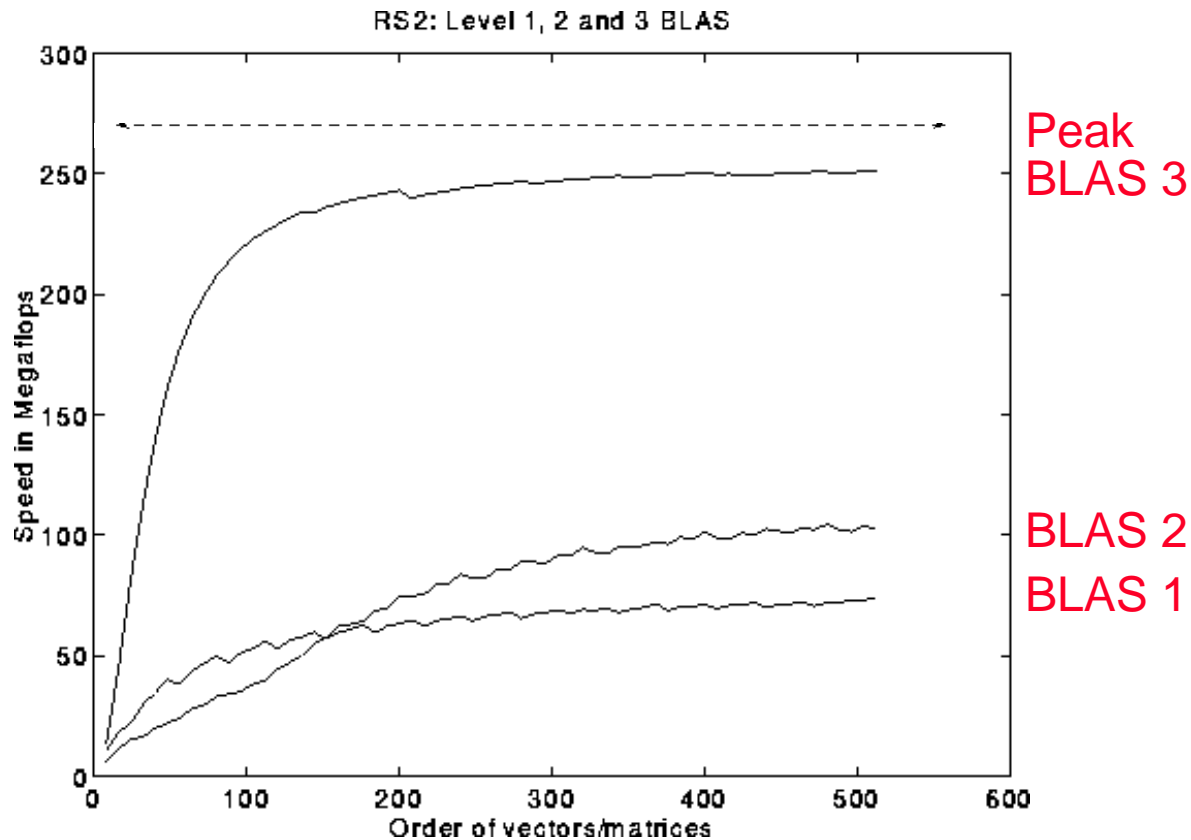
# Basic Linear Algebra Subroutines

---

- 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)
  - See [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/lapack](http://www.netlib.org/lapack)

# BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops



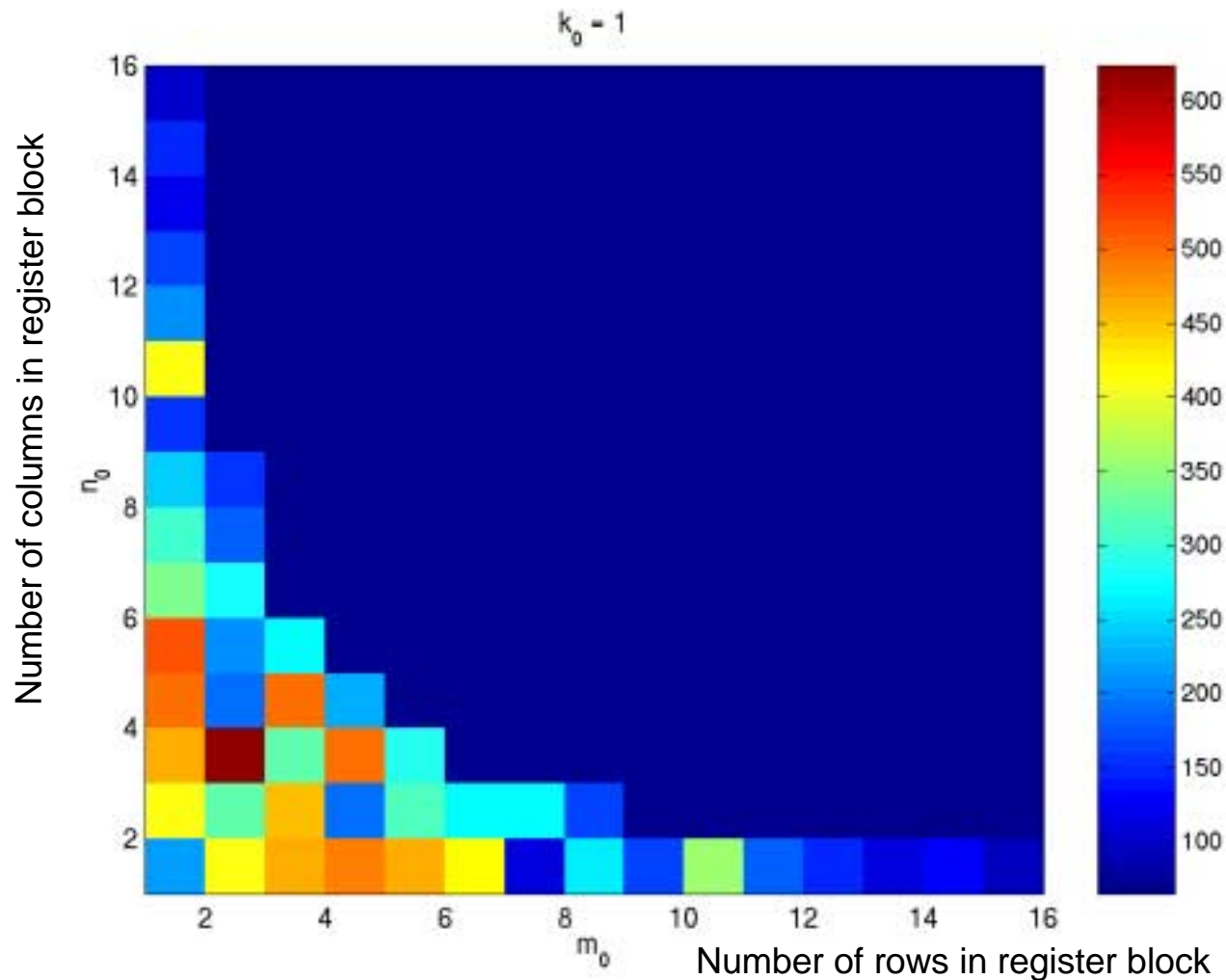
BLAS 3 (n-by-n matrix matrix multiply) vs  
BLAS 2 (n-by-n matrix vector multiply) vs  
BLAS 1 (saxpy of n vectors)

# 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.
- Some systems use search
  - Atlas – being incorporated into Matlab
  - BeBOP – <http://www.cs.berkeley.edu/~richie/bebop>

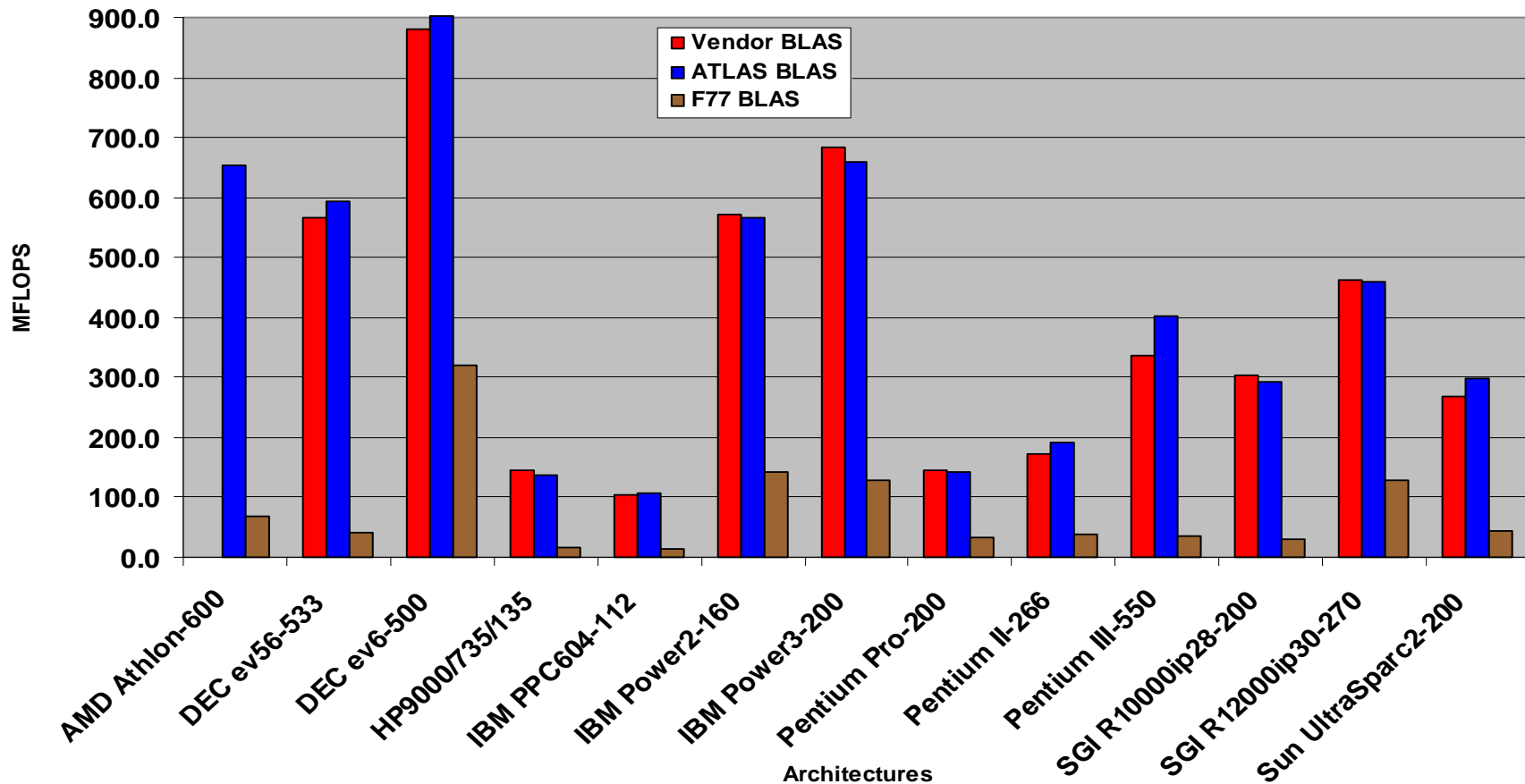
# What the Search Space Looks Like



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 $n = 500$ )

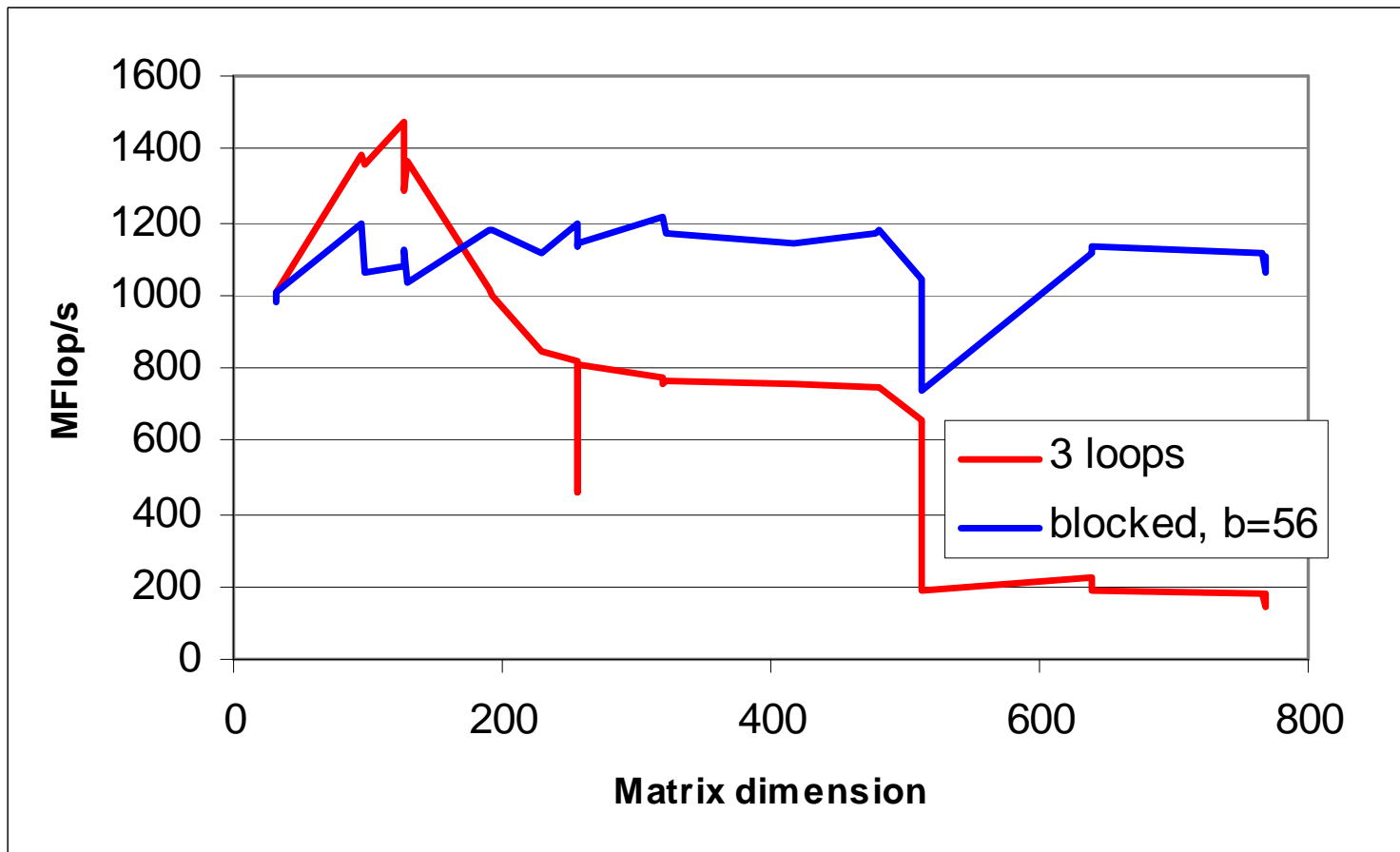
Source: Jack Dongarra



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

# 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 hw1



# 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
  - BeBOP: [bebop.cs.berkeley.edu/](http://bebop.cs.berkeley.edu/)
  - 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;           false read-after-write hazard  
a[i+1] = b[i+1] * d;     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

# 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

# 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

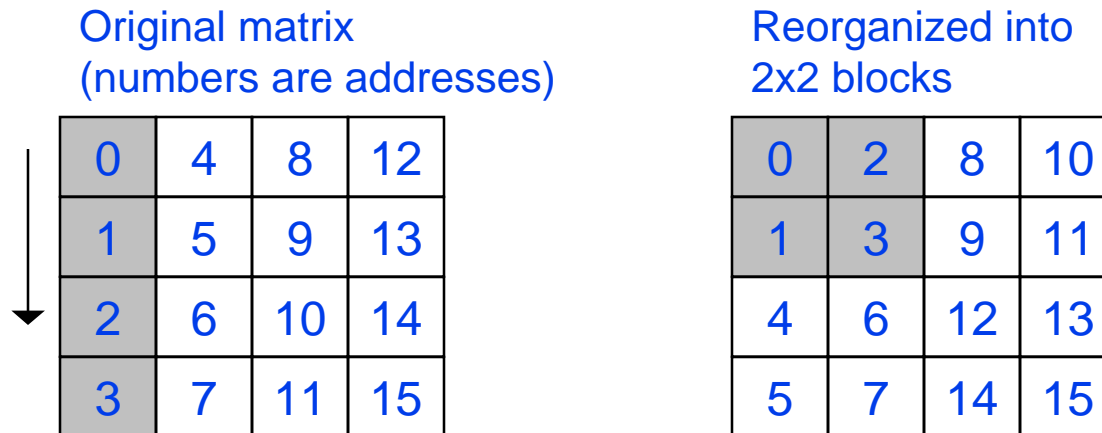
- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

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

# Copy optimization

---

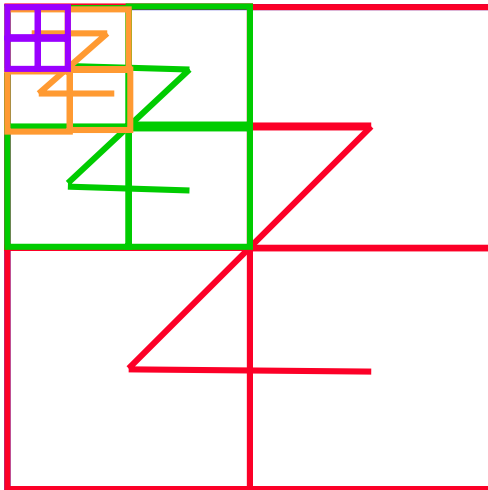
- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality



# Recursive Data Layouts

---

- Copy optimization often works because it improves spatial locality
- 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
- 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

# 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  $2 \times 2$  matrix multiply, normally takes 8 multiplies, 7 adds
  - Strassen does it with 7 multiplies and 18 adds

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

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

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

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

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

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

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

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

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

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

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

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

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

# 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
  - 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:
      - $q = f/m = \# \text{ floating point opns} / \# \text{ slow memory opns}$
    - 2.machine balance in the memory system:
      - $t_m/t_f = \text{time for slow memory operation} / \text{time for floating point operation}$
- 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 1
  - Work in teams of 2 or 3 (assigned this time)

# Reading for Today

---

- 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.

## 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. What is LAPACK? ScaLAPACK?
7. Why does loop unrolling improve uniprocessor performance?

# Reading Assignment

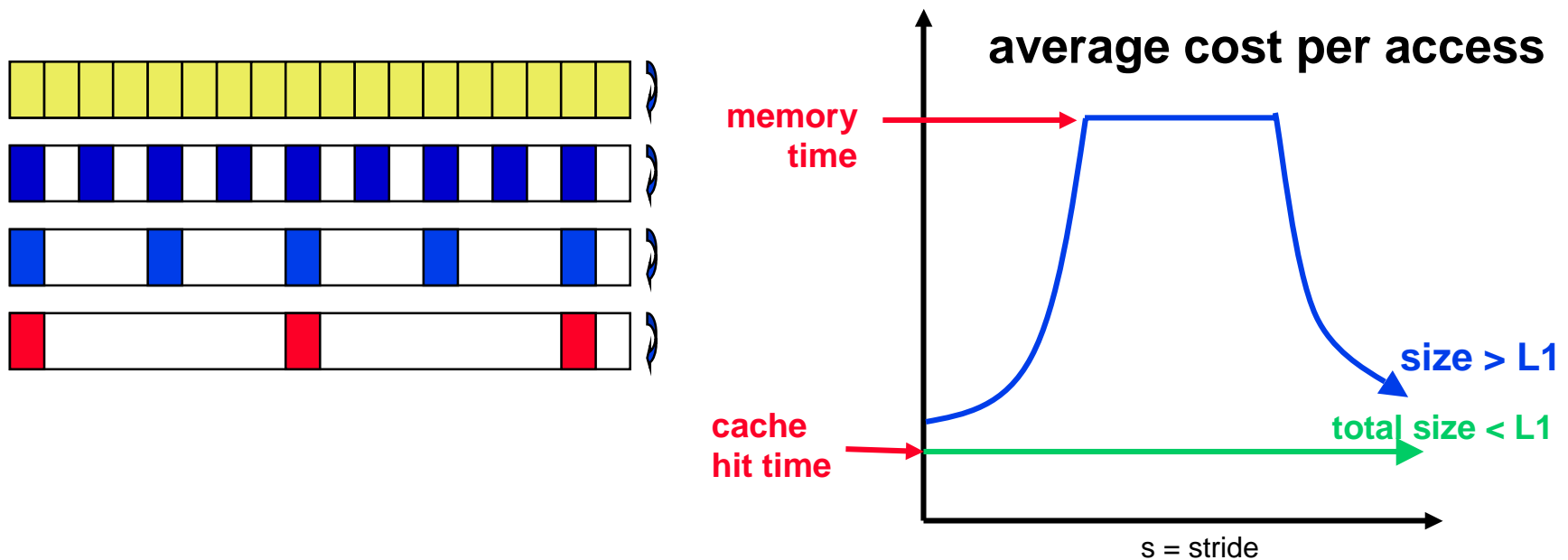
---

- Next week: Current high performance architectures
  - Cray X1
    - <http://www.sc-conference.org/sc2003/paperpdfs/pap183.pdf>
  - Blue Gene L
    - <http://sc-2002.org/paperpdfs/pap.pap207.pdf>
  - Clusters
    - <http://www.mirror.ac.uk/sites/www.beowulf.org/papers/ICPP95/>
- Optional
  - Chapter 1,2 of the “Sourcebook of Parallel Computing”
  - Alternative to Beowulf paper:
    - <http://now.cs.berkeley.edu/Case/case.html>

---

# Review from Last Lecture

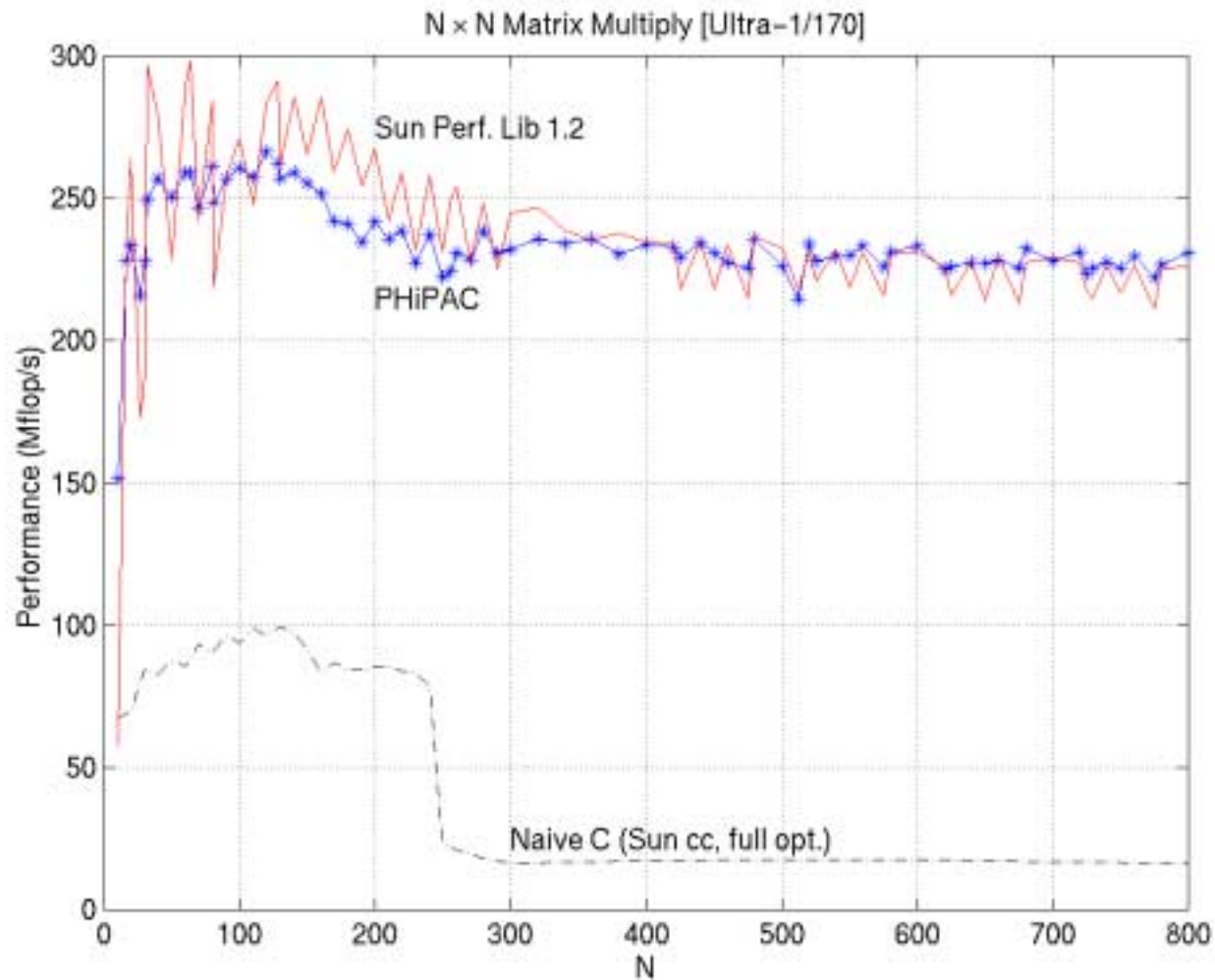
# Membench: What to Expect



- Consider the average cost per load
  - Plot one line for each array size, 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



# PHiPAC: Portable High Performance ANSI C



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