Optimizing Cache Performance in Matrix Multiplication
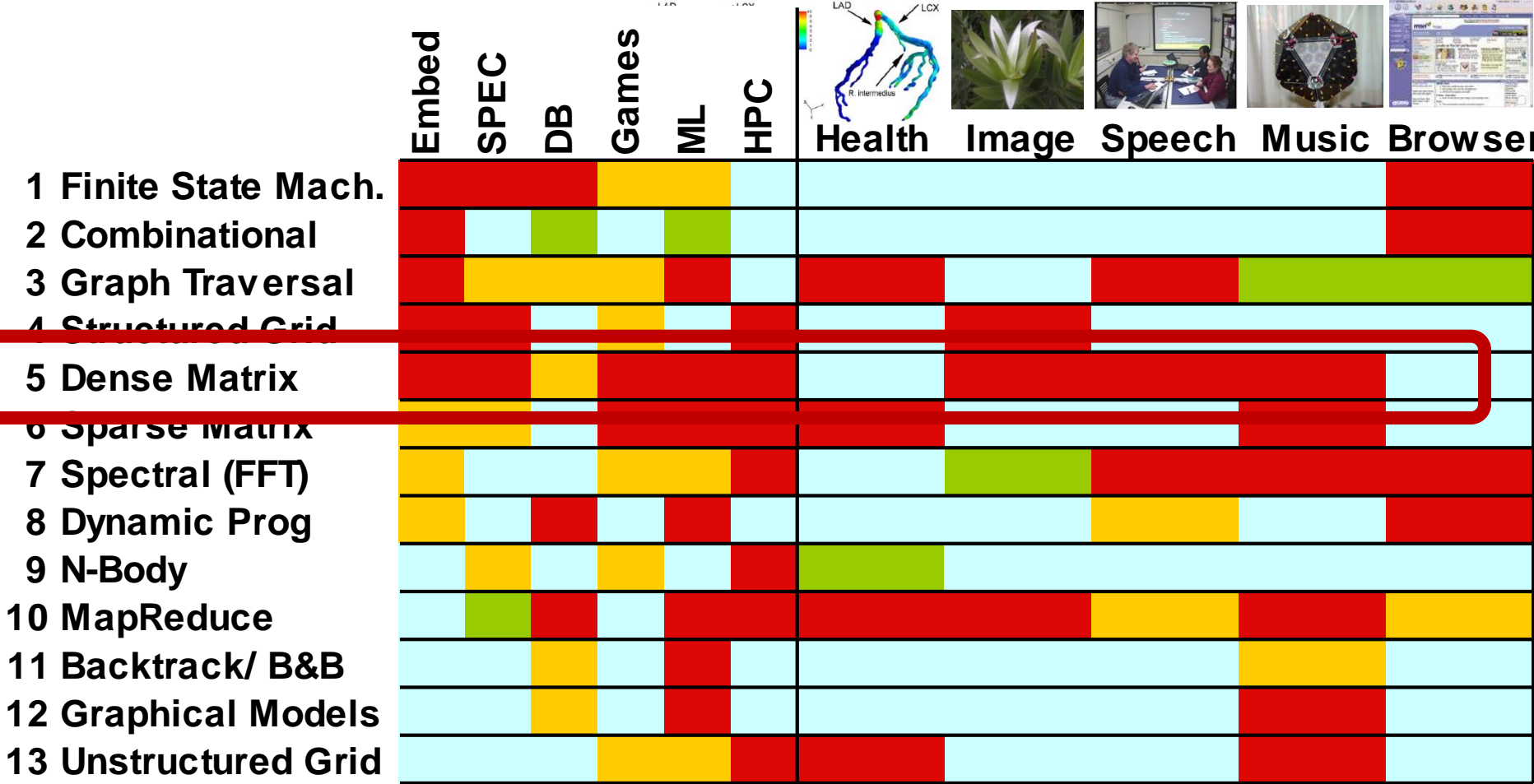
UCSB CS240A, Winter 2016

Modified from Demmel/Yelick's slides

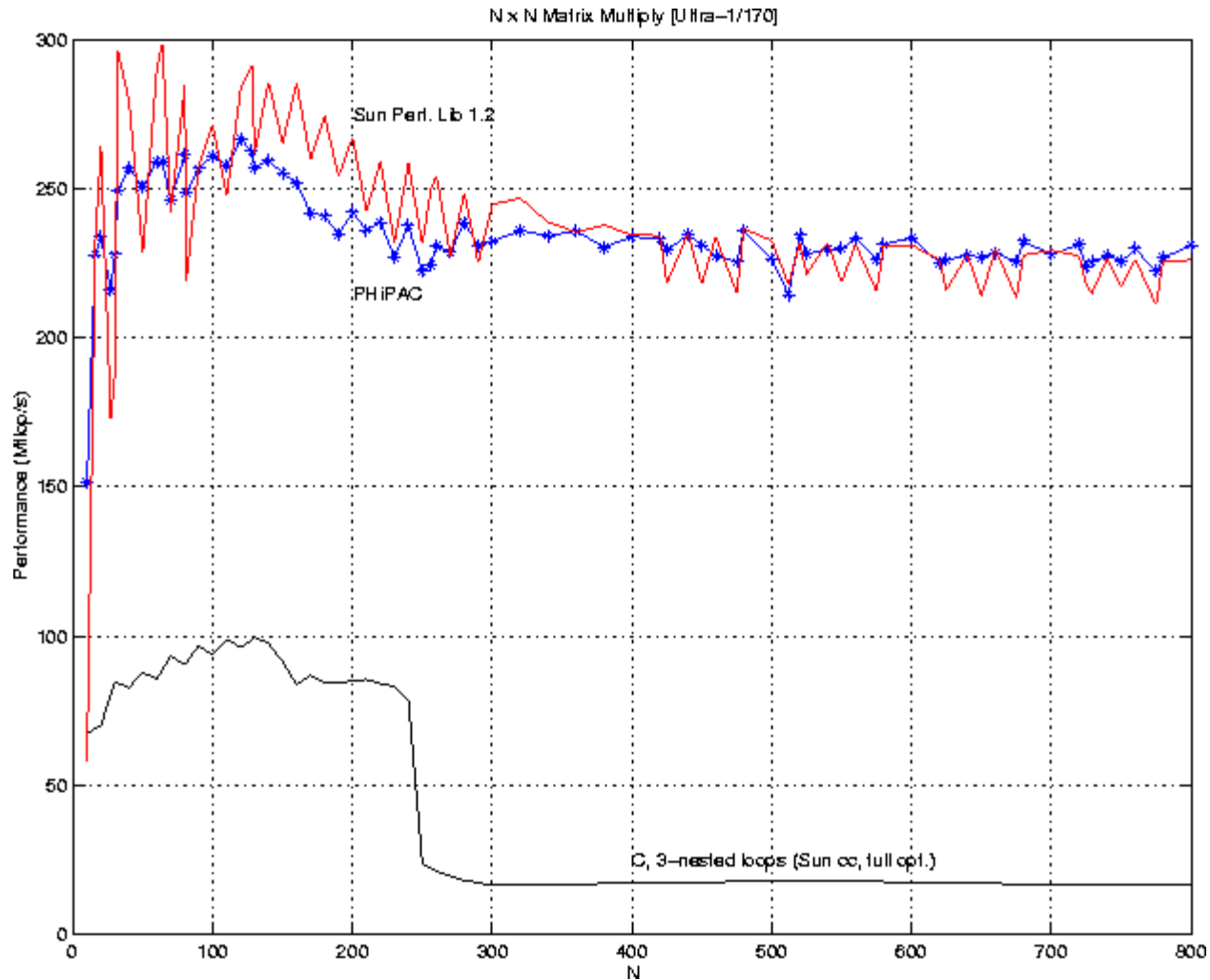# Case Study with Matrix Multiplication

- An important kernel in many problems

- Optimization ideas can be used in other problems
  - The most-studied algorithm in high performance computing

- How to measure quality of implementation in terms of performance?

  - Megaflops number

  - Defined as: Core computation count / time spent

  - Matrix-matrix multiplication operation count = 2 n^3

- Example: 300MFLOPS → 300 million MM-related floating operations performed per second.

What do commercial and CSE applications have in common?

(Red Hot → Blue Cool)

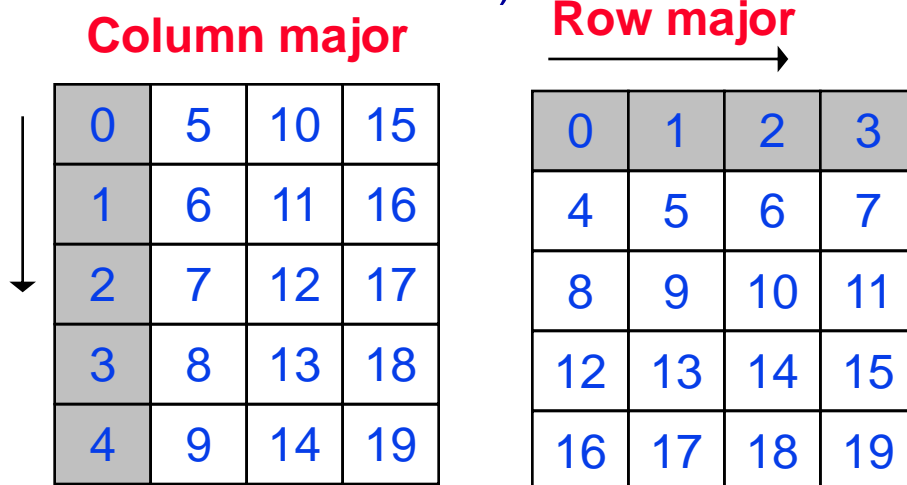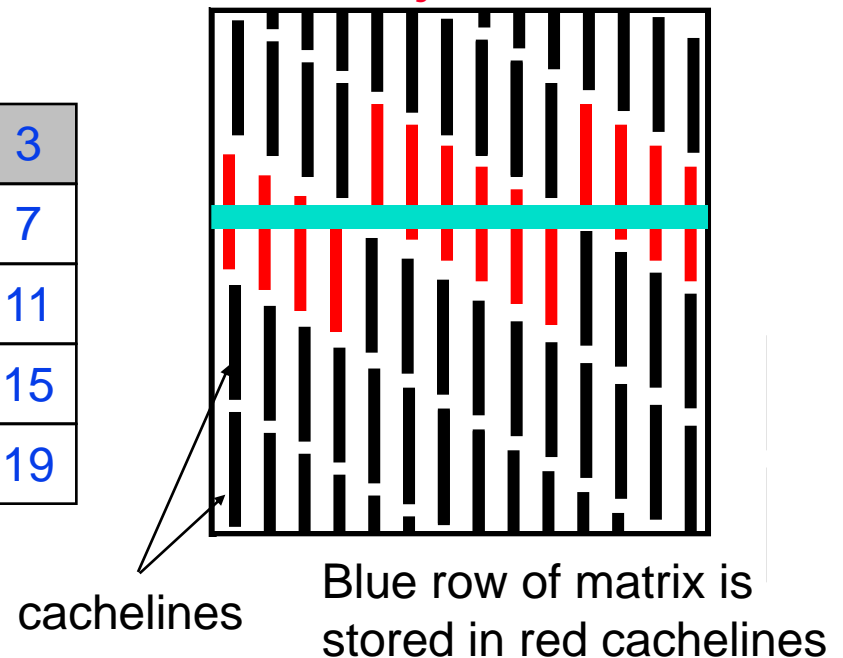|	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser
1 Finite State Mach.											
2 Combinational											
3 Graph Traversal											
4 Structured Grid											
5 Dense Matrix											
6 Sparse Matrix											
7 Spectral (FFT)											
8 Dynamic Prog											
9 N-Body											
10 MapReduce											
11 Backtrack/ B&B											
12 Graphical Models											
13 Unstructured Grid											

Matrix-multiply, optimized several ways

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

Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D"

- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A(i,j) at A+i*n+j
  - recursive later)

Column major

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

Row major

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

Column major matrix in memory

cachelines

Blue row of matrix is stored in red cachelines

- Column major (for now)

Figure source: Larry Carter, UCSD 5

Using a Simple Model of Memory to Optimize

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation $<< t_m$
  - $q = f / m$ average number of flops per slow memory access

Computational Intensity: **Key to algorithm efficiency**

- Minimum possible time = $f * t_f$ when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$

Machine Balance: **Key to machine efficiency**

- 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

{implements y = y + A*x}

for i = 1 to n

     for j = 1 to n

          y(i) = y(i) + A(i,j)*x(j)
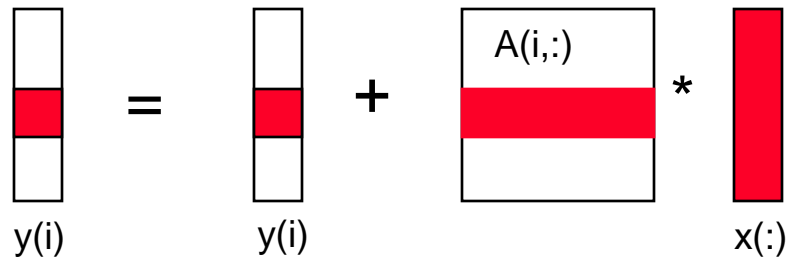
Warm up: Matrix-vector multiplication

{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1 to n
    {read row i of A into fast memory}
    for j = 1 to 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 = $f / m \approx 2$
- Time

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

- Megaflop rate = f/ Time = $1 / (t_f + 0.5\, t_m)$
- Matrix-vector multiplication limited by slow memory speed

Modeling Matrix-Vector Multiplication

- Compute time for nxn = 1000x1000 matrix
- For t_f and t_m, using data from R. Vuduc's PhD (pp 351-3)
  - http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf
  - For t_m use minimum-memory-latency / words-per-cache-line

|	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

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

Simplifying Assumptions

- What simplifying assumptions did we make in this analysis?
    - Ignored parallelism in processor between memory and arithmetic within the processor
        - Sometimes drop arithmetic term in this type of analysis
    - Assumed fast memory was large enough to hold three vectors
        - Reasonable if we are talking about any level of cache
        - Not if we are talking about registers (~32 words)
    - Assumed the cost of a fast memory access is 0
        - Reasonable if we are talking about registers
        - Not necessarily if we are talking about cache (1-2 cycles for L1)
    - Memory latency is constant
- Could simplify even further by ignoring memory operations in X and Y vectors
- Megaflop rate = $1 / (t_f + 0.5 t_m)$

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
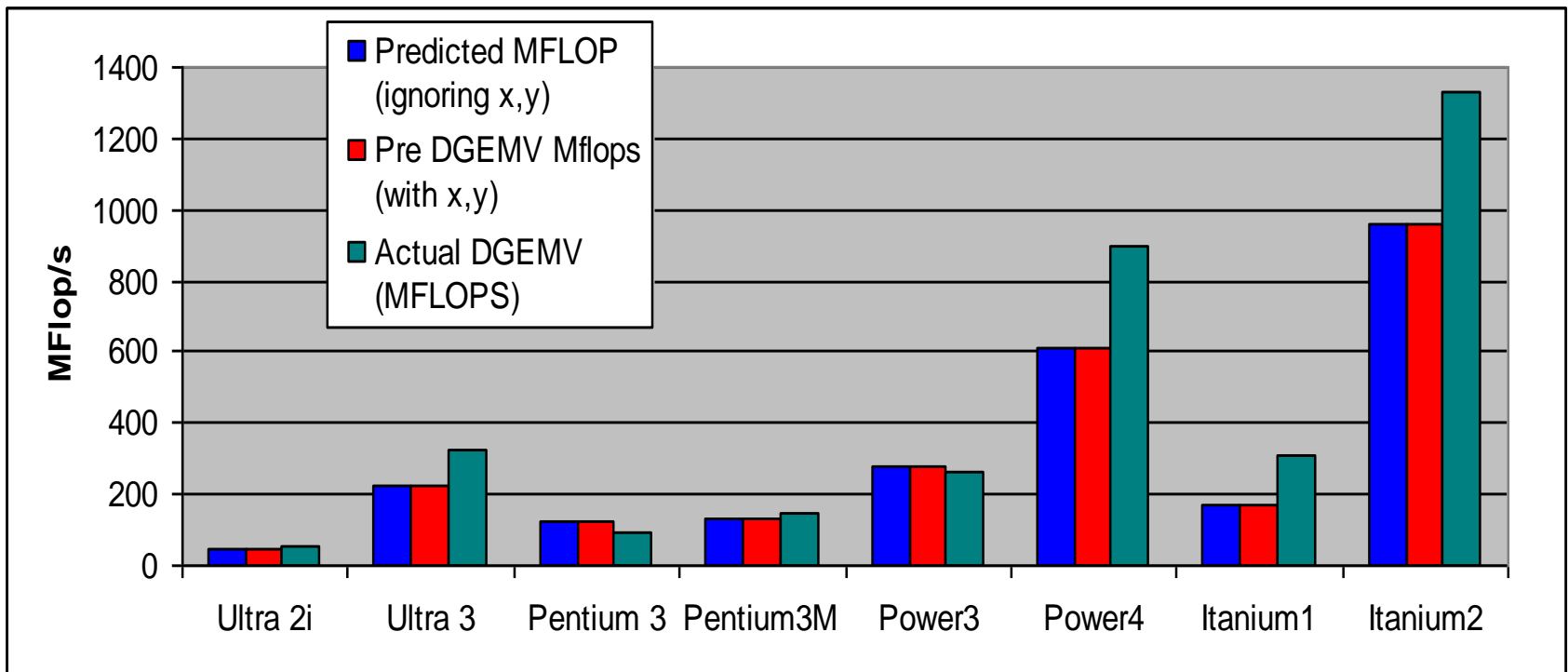
Naïve Matrix-Matrix Multiplication
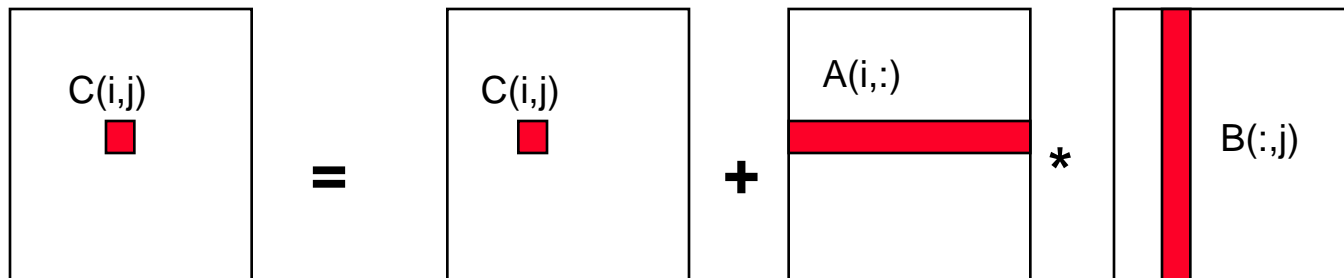
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)

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

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

Naïve Matrix Multiply

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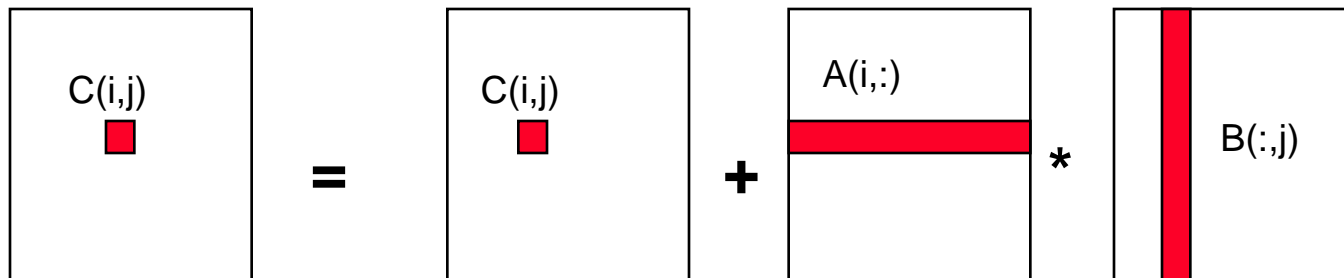 for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
   {write C(i,j) back to slow memory}

Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

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

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

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

$= n^3 + 3n^2$

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

≈ 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
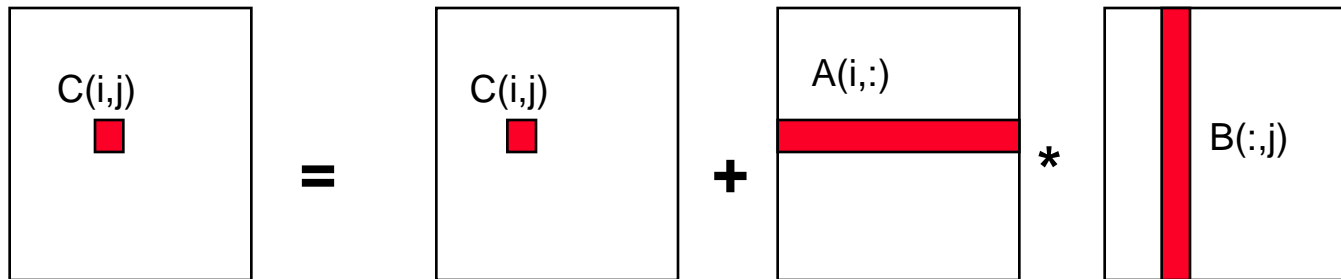
Partitioning for blocked matrix multiplication

- Example of submartix partitioning

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right) \implies \left(\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right)$$

$$A_{11} = \left(\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right), A_{12} = \left(\begin{array}{cc} a_{13} & a_{14} \\ a_{23} & a_{24} \end{array} \right)$$

$$A_{21} = \left(\begin{array}{cc} a_{31} & a_{32} \\ a_{41} & a_{42} \end{array} \right), A_{22} = \left(\begin{array}{cc} a_{33} & a_{34} \\ a_{43} & a_{44} \end{array} \right)$$
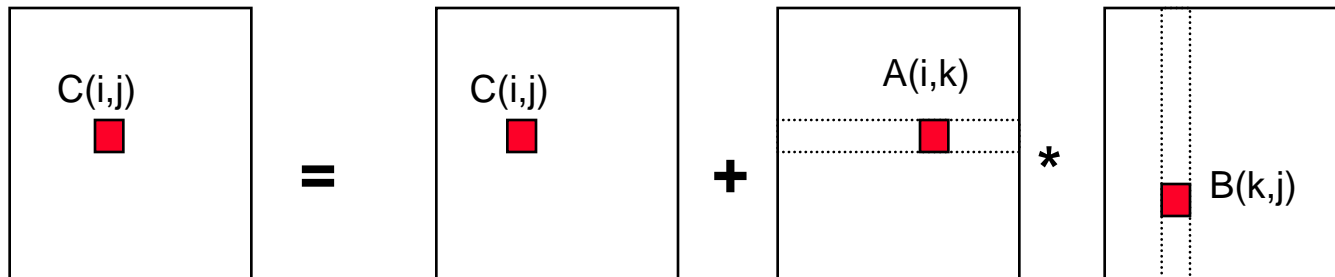
Blocked (Tiled) Matrix Multiply

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
                for k = 1 to N
                        C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}

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

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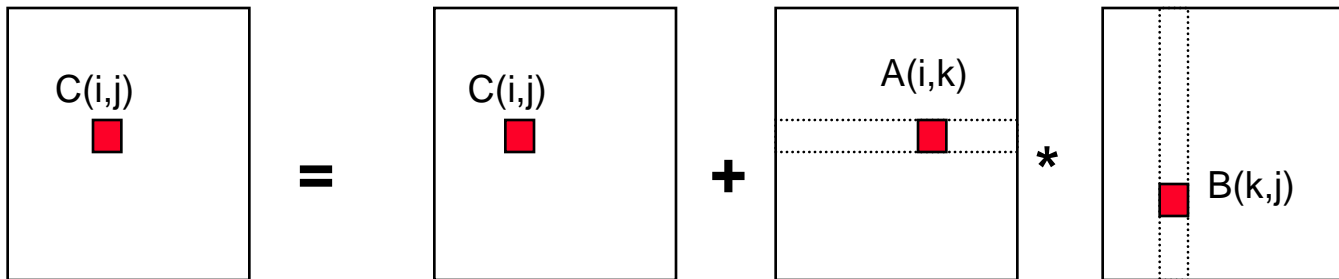   for k = 1 to N
              {read block A(i,k) into fast memory}
              {read block B(k,j) into fast memory}
              C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
          {write block C(i,j) back to slow memory}

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

Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between slow and fast memory

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

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

q = f / m is our measure of memory access efficiency

So:

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

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

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

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

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

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

So we can improve performance by increasing the blocksize b

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

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}

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

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

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

- This size is reasonable for L1 cache, but not for register sets

- Note: analysis assumes it is possible to schedule the instructions perfectly

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

Basic Linear Algebra Subroutines (BLAS)

- **Industry standard interface (evolving)**
    - **www.netlib.org/blas,   www.netlib.org/blas/blast--forum**
- **Vendors, others supply optimized implementations**
- **History**
    - **BLAS1 (1970s):**
        - **vector operations: dot product, saxpy (y=α*x+y), etc**
        - **m=2*n, f=2*n, q ~1 or less**
    - **BLAS2 (mid 1980s)**
        - **matrix-vector operations. Example: matrix vector multiply, etc**
        - **m=n^2, f=2*n^2, q~2, less overhead**
        - **somewhat faster than BLAS1**
    - **BLAS3 (late 1980s)**
        - **matrix-matrix operations: Example: matrix matrix multiply, etc**
        - **m <= 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 used BLAS3 when possible (LAPACK & ScaLAPACK)**
    - **See `www.netlib.org/{lapack,scalapack}`**
    - **`If BLAS3 is not possible, use BLAS2 if applicable. Otherwise BLAS1.`**

BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

RS2: Level 1, 2 and 3 BLAS
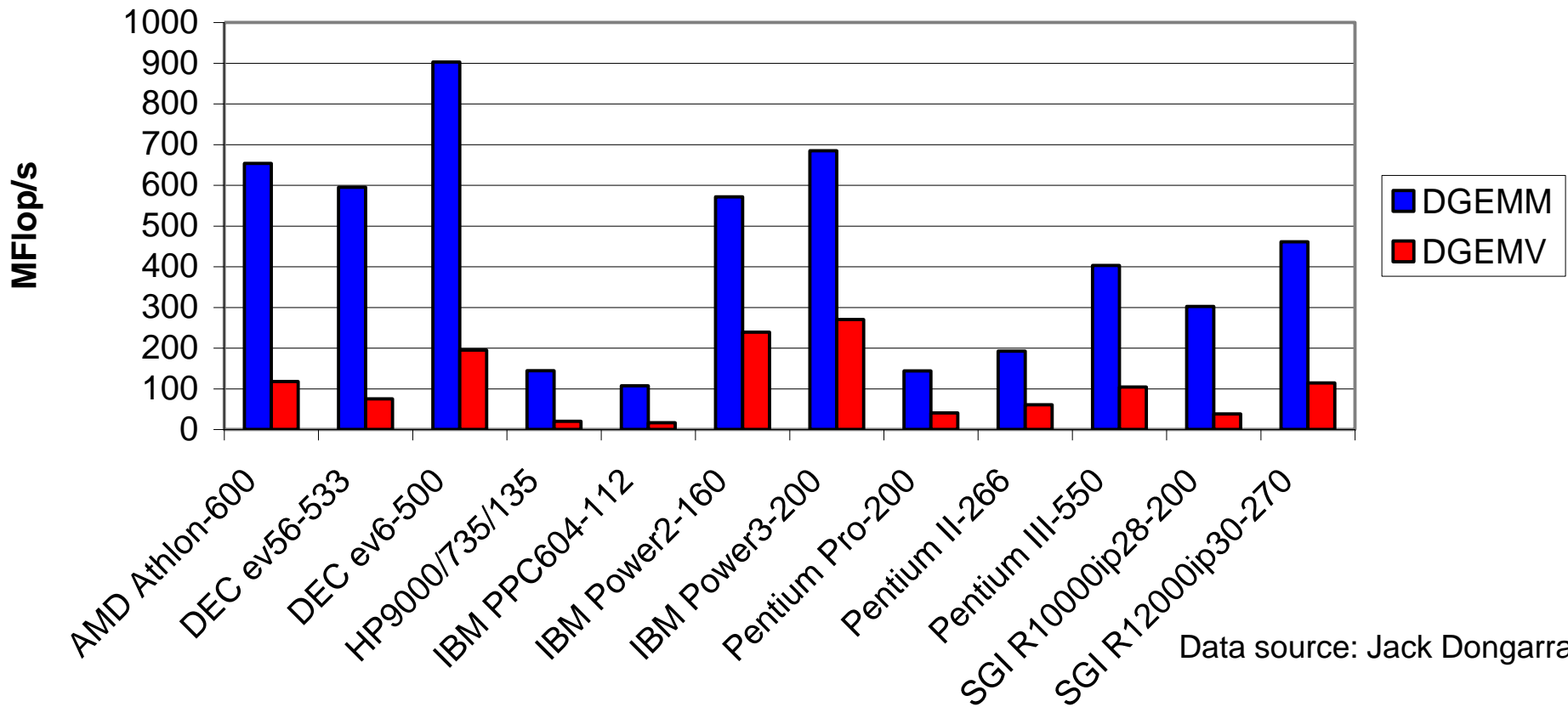

Peak
BLAS 3
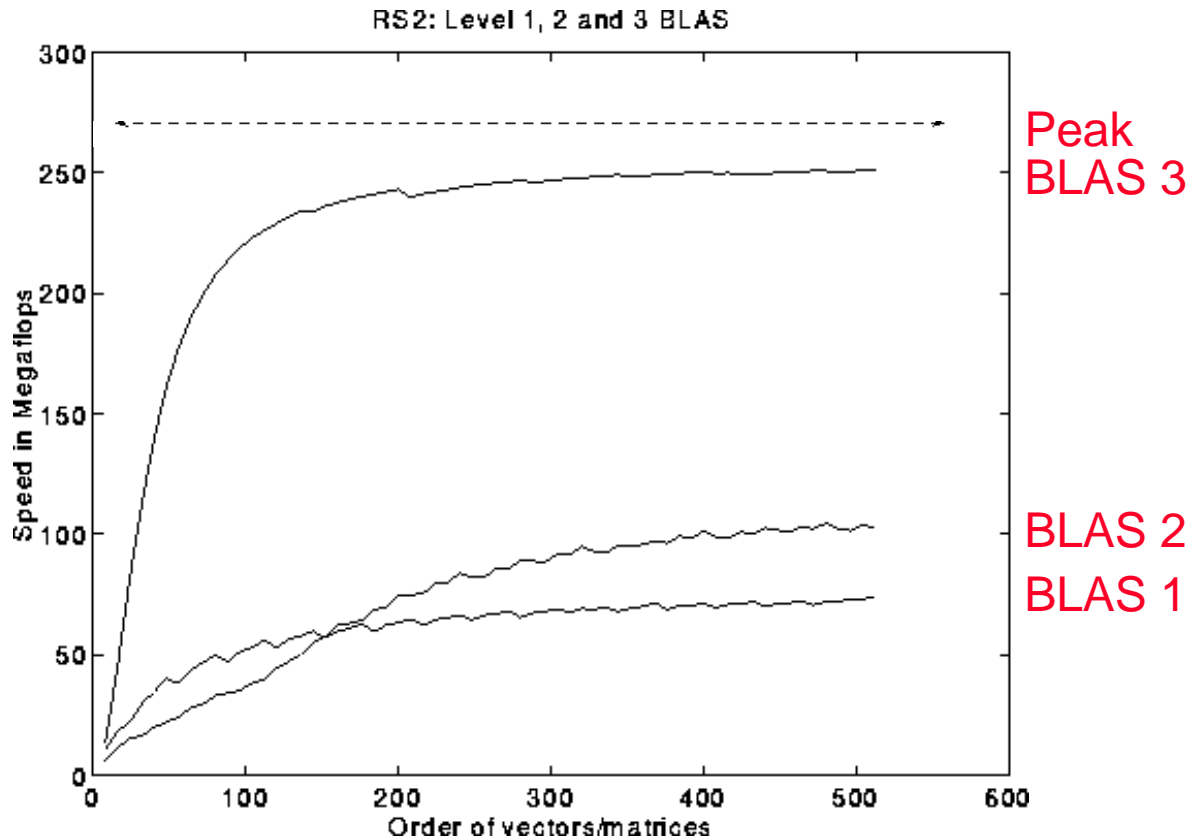
BLAS 2
BLAS 1

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)

Data source: Jack Dongarra

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:

     - $q = f/m$ = # floating point operations / # slow memory references

  2. machine balance in the memory system:

     - t_m/t_f = time for slow memory reference / 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

Questions You Should Be Able to Answer

1. What is the key to understand algorithm efficiency in our simple memory model?

2. Why does block matrix multiply reduce the number of memory references?

   2D blocking is sometime called tiling

3. What are the BLAS?

Summary

- 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
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby that recently used
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Can rearrange code/data to improve locality
  - Useful techniques: Blocking. Loop exchange.