

# **CSC367 Parallel computing**

## **Lecture 3: Single Processor Machines-Performance Model**

Maryam Mehri Dehnavi  
mmehride@cs.toronto.edu

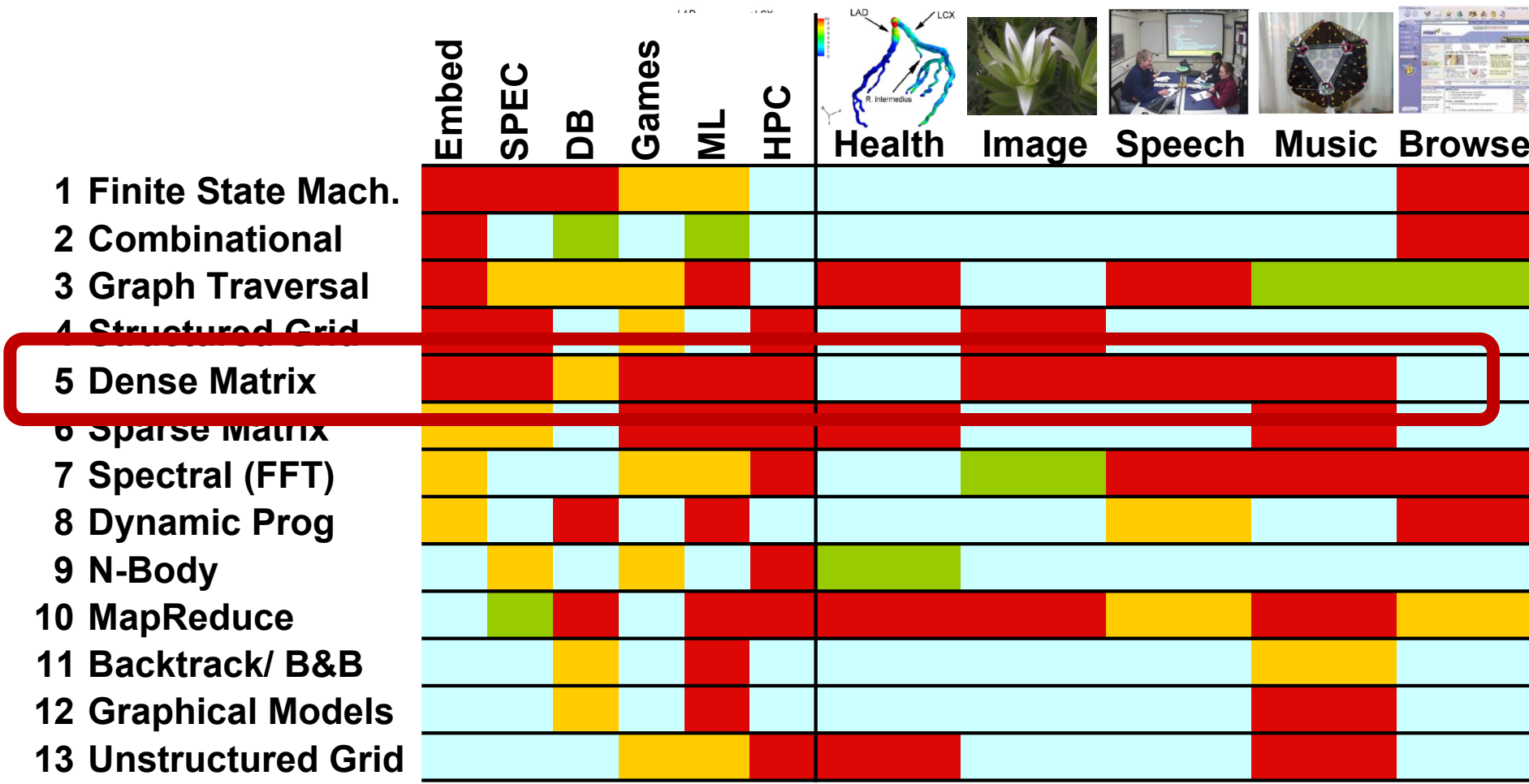
# Outline

## A performance model for 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
- BLAS routines

# What do commercial and CSE applications have in common?


## Motif/Dwarf: Common Computational Methods (Red Hot → Blue Cool)



# 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

**Column major**



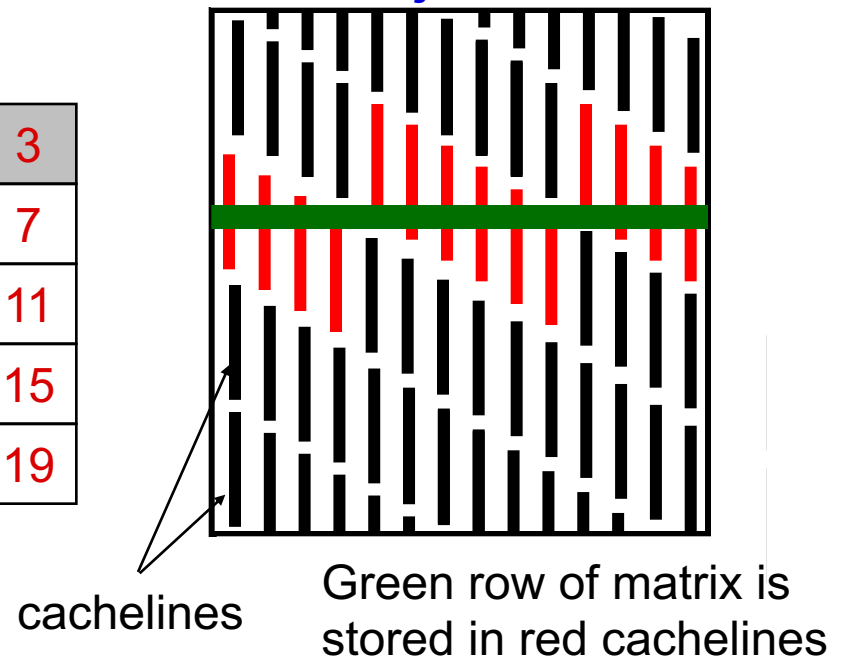
0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

**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
  - $f * t_f + m * t_m = f * t_f * (1 + \frac{t_m}{t_f} * 1/q)$
- 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

architecture dependent

algo specific

**Machine Balance:** Key to machine efficiency

often overlooked ... but this is the part that can be optimized

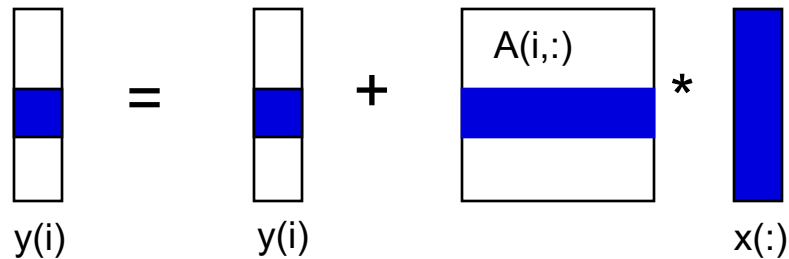
# Warm up: Matrix-vector multiplication

```
{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

slow memory refs: read/write to/from slow memory

{read x(1:n) into fast memory}	n
{read y(1:n) into fast memory}	n
for i = 1:n	
{read row i of A into fast memory}	$n^2$
for j = 1:n	
$y(i) = y(i) + A(i,j)*x(j)$	
{write y(1:n) back to slow memory}	n

assume: fast cache able to fit 1 col of A, x, y



- $m$  = 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

- Examples of some architectures and their machine balance
- So the computational intensity of 2 in matrix-vector multiply means that we can not get close to half peak of these machines: Memory bound operation!

	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)*

memory bound -> cannot optimize ...



# Naïve Matrix Multiply

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

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

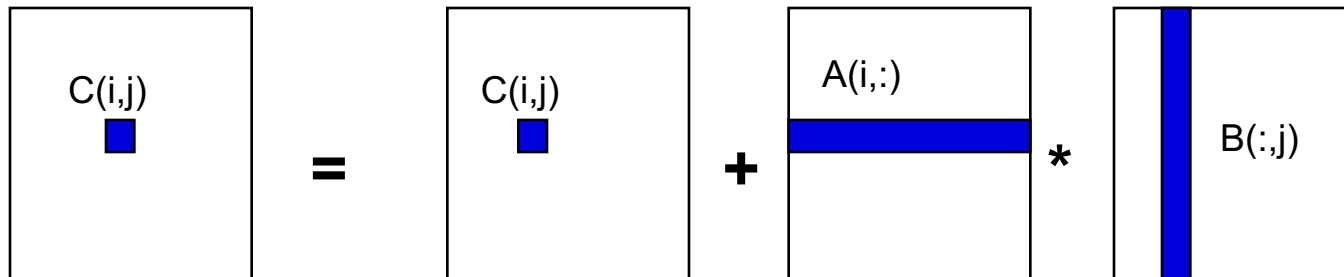
```
    for j = 1 to n
```

```
        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  $2*n^3 / 3*n^2 = O(n)$



Assume for simplicity that data is layed out in slow memory in the order it is being accessed. For example, here  $A$  is stored in row-major and  $B$  is stored in column-major.

# Naïve Matrix Multiply

loopiter(#access inside 1 iter)

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

for  $i = 1$  to  $n$

{read row  $i$  of  $A$  into fast memory}  $n(n)$

for  $j = 1$  to  $n$

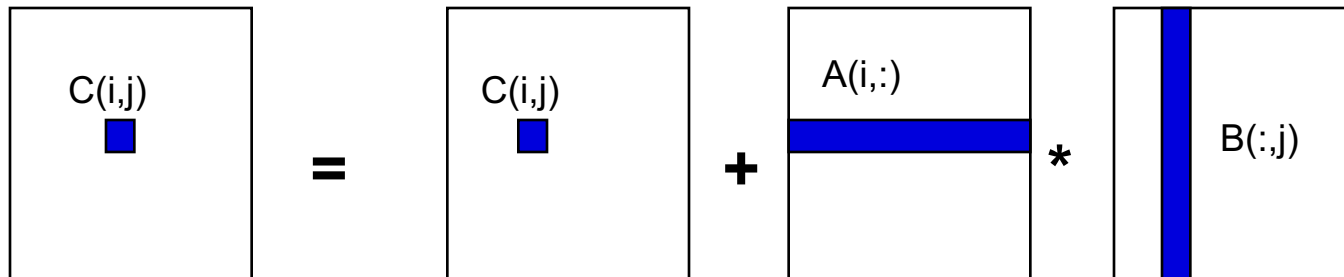
{read  $C(i,j)$  into fast memory}  $n^2(1)$

{read column  $j$  of  $B$  into fast memory}  $n^2(n)$

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}  $n^2(1)$



# Naïve Matrix Multiply

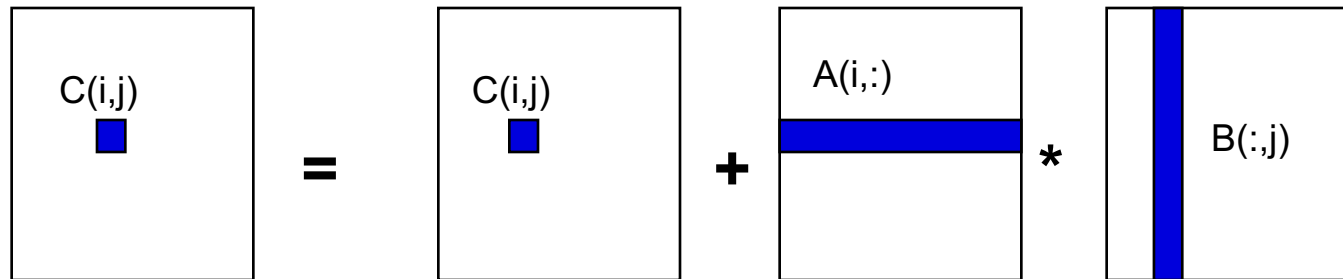
Number of slow memory references on unblocked matrix multiply

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

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

$\approx 2$  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



# 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}

**cache does this automatically**

**3 nested loops inside**

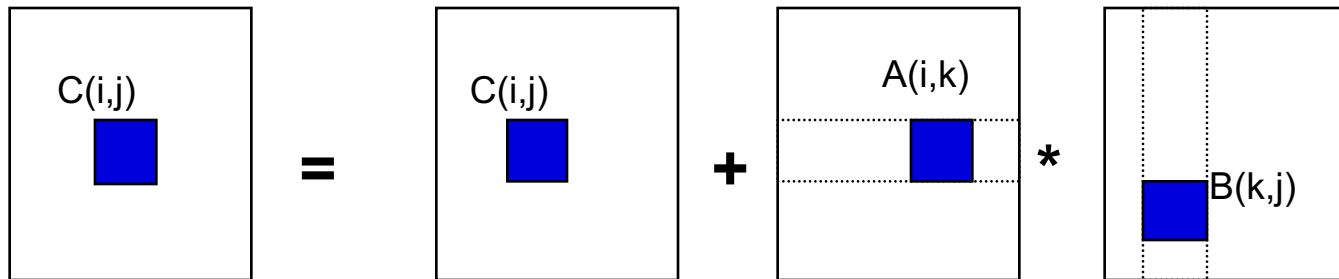
**block size = loop bounds**

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

$$N^3 * (b^2)$$

$$N^3 * (b^2)$$

$$N^2 * (b^2)$$



Tiling for registers (managed by you/compiler) or caches (hardware)

# Blocked (Tiled) Matrix Multiply

Recall:

$N$  bxb submatrices

$m$  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:

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

So computational intensity  $q = ?$

# 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:

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

So computational intensity  $q = f / m = 2n^3 / ((2N + 2) * n^2)$   
 $\approx n / N = b$  for large  $n$

So we can improve performance by increasing the blocksize  $b$

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

# 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:

$m = N * n^2$  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

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

$b$  limited by fast cache size

**Why not increase**

So computational intensity  $q = f / m = 2n^3 / ((2N + 2) * n^2)$  **b to a very large**

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

**number?**

So we can improve performance by increasing the blocksize  $b$

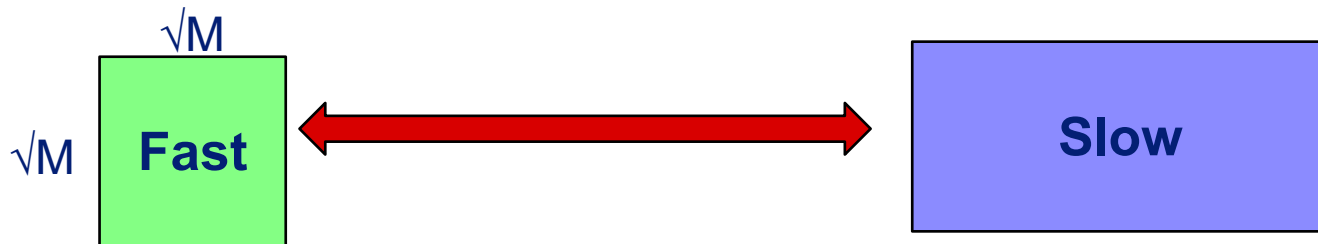
Can be much faster than matrix-vector multiply ( $q=2$ )

# Limits to Optimizing Matrix Multiply

3 submatrices of A,B,C

- The tiled matrix multiply analysis assumes that three tiles/blocks fit into fast memory at once.
- If  $M_{\text{fast}}$  is the size of fast memory then the previous analysis showed that the blocked algorithm has computational intensity:

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





# 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): 15 different operations
    - vector operations: dot product, saxpy ( $y = \alpha * x + y$ ), etc
    - $m = 2 * n$ ,  $f = 2 * n$ ,  $q = f / m =$  computational intensity  $\sim 1$  or less

# Basic Linear Algebra Subroutines (BLAS)

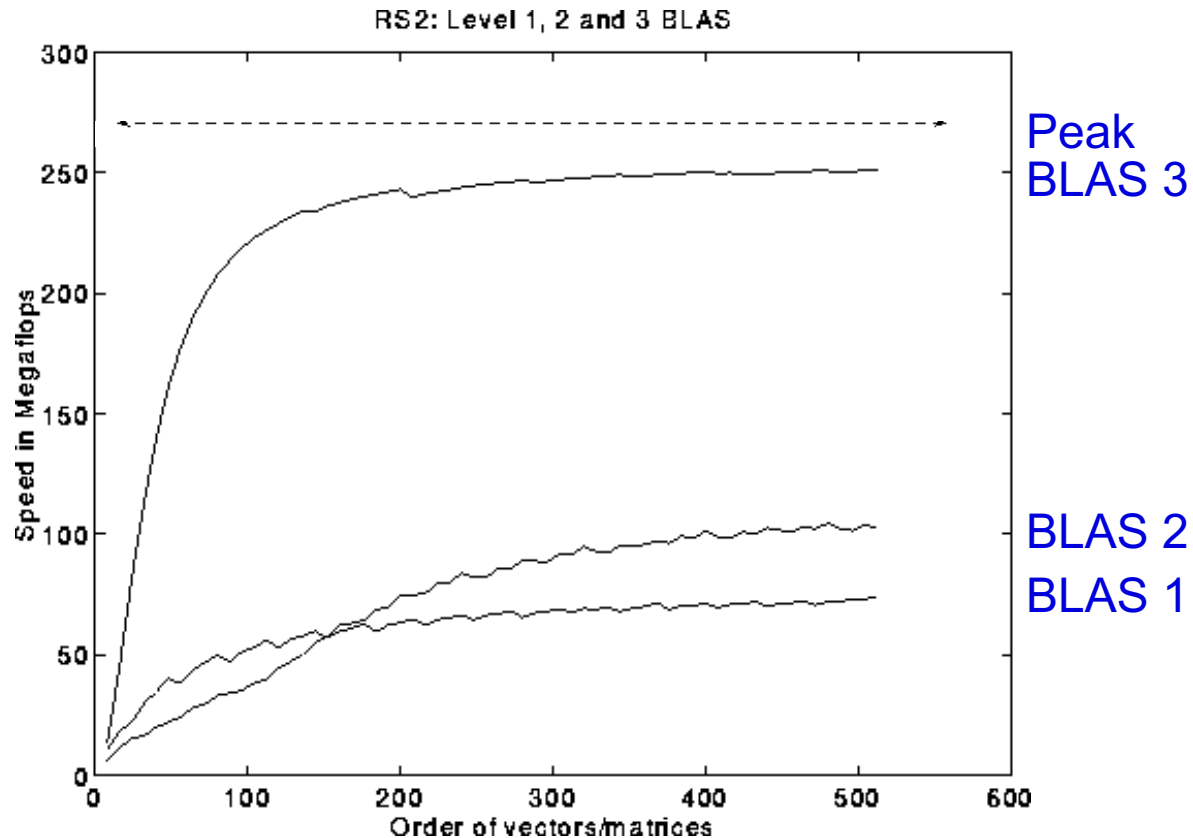
- Industry standard interface (evolving)
  - [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/blas/blast--forum](http://www.netlib.org/blas/blast--forum)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s): 15 different operations
    - vector operations: dot product, saxpy ( $y = \alpha * x + y$ ), etc
    - $m = 2 * n$ ,  $f = 2 * n$ ,  $q = f / m =$  computational intensity  $\sim 1$  or less
  - BLAS2 (mid 1980s): 25 different operations
    - matrix-vector operations: matrix vector multiply, etc
    - $m = n^2$ ,  $f = 2 * n^2$ ,  $q \sim 2$ , less overhead
    - somewhat faster than BLAS1

# 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): 15 different operations
    - vector operations: dot product, saxpy ( $y = \alpha * x + y$ ), etc
    - $m = 2 * n$ ,  $f = 2 * n$ ,  $q = f / m =$  computational intensity  $\sim 1$  or less
  - BLAS2 (mid 1980s): 25 different operations
    - 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): 9 different operations
    - 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})

# 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)

