High Performance Matrix Computations: case study: matrix multiplications and BLAS

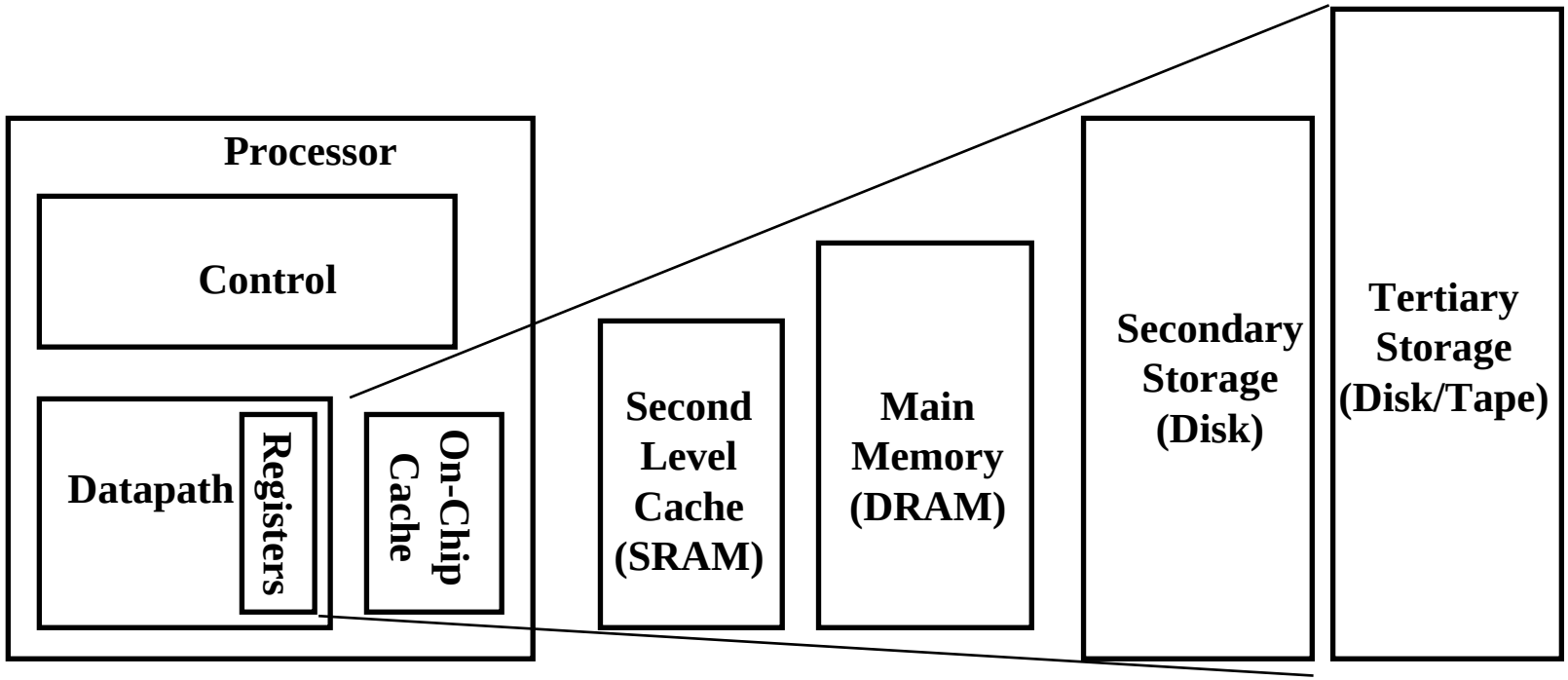
Outline

1. Memory Hierarchies
2. Cache and its importance in performance
3. Optimizing matrix multiply for caches
4. BLAS
5. Bag of Tricks
6. Supplement: Strassen's algorithm

Memory Hierarchy

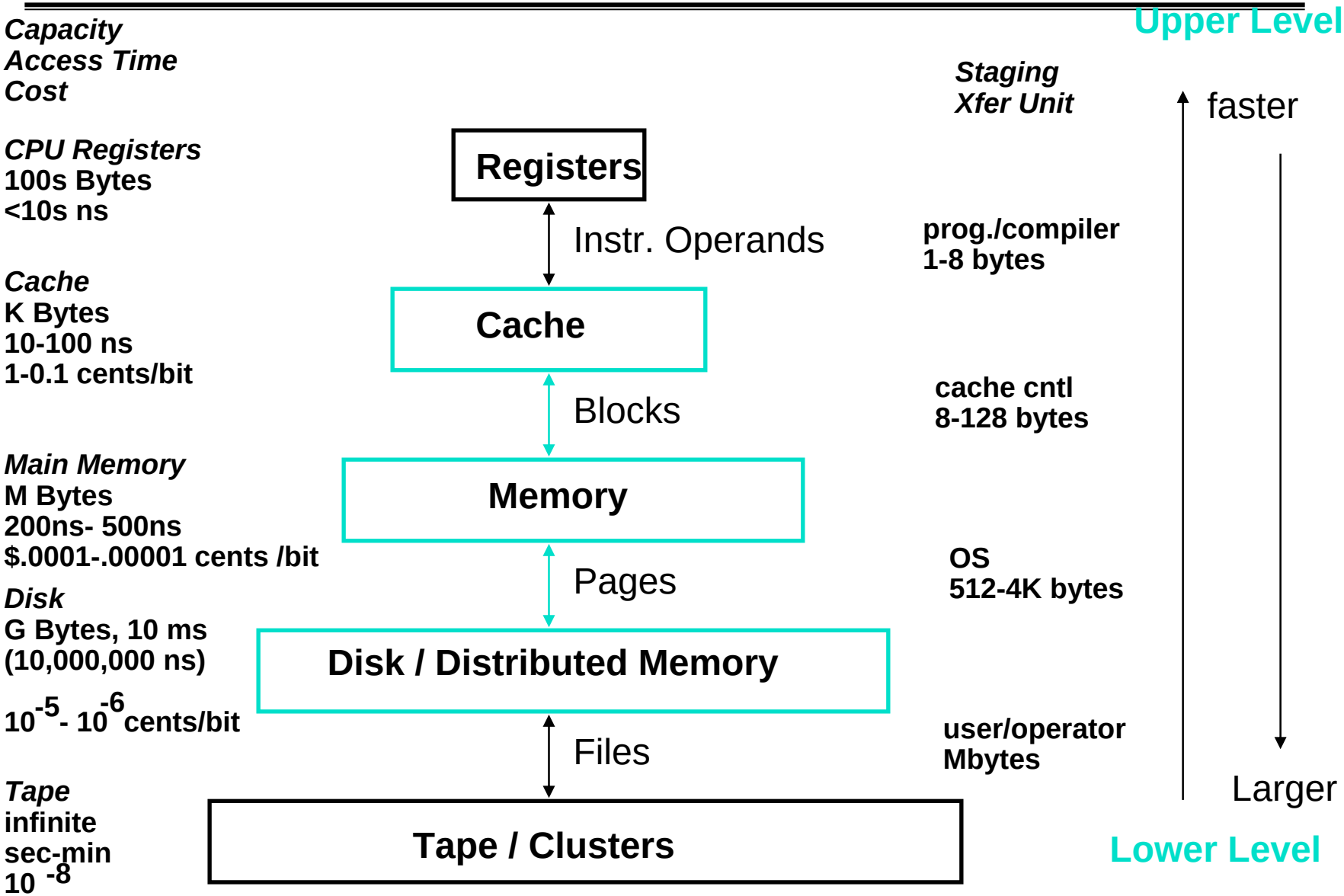
- Most programs have a high degree of locality in their accesses
  - spatial locality: accessing things nearby previous accesses
  - temporal locality: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality
- By taking advantage of the principle of locality:
  - present the user with as much memory as is available in the cheapest technology
  - Provide access at the speed offered by the fastest technology

Memory Hierarchy



Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

Levels of the Memory Hierarchy



Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
  - Exist in the program stack, static region, or heap
- Operations include
  - Read and write (given an address/pointer)
  - Arithmetic and other logical operations
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into “obvious” lower level instructions
  - Hardware executes instructions in order specified by compiler
- Cost
  - Each operations has roughly the same cost (read, write, add, multiply, etc.)

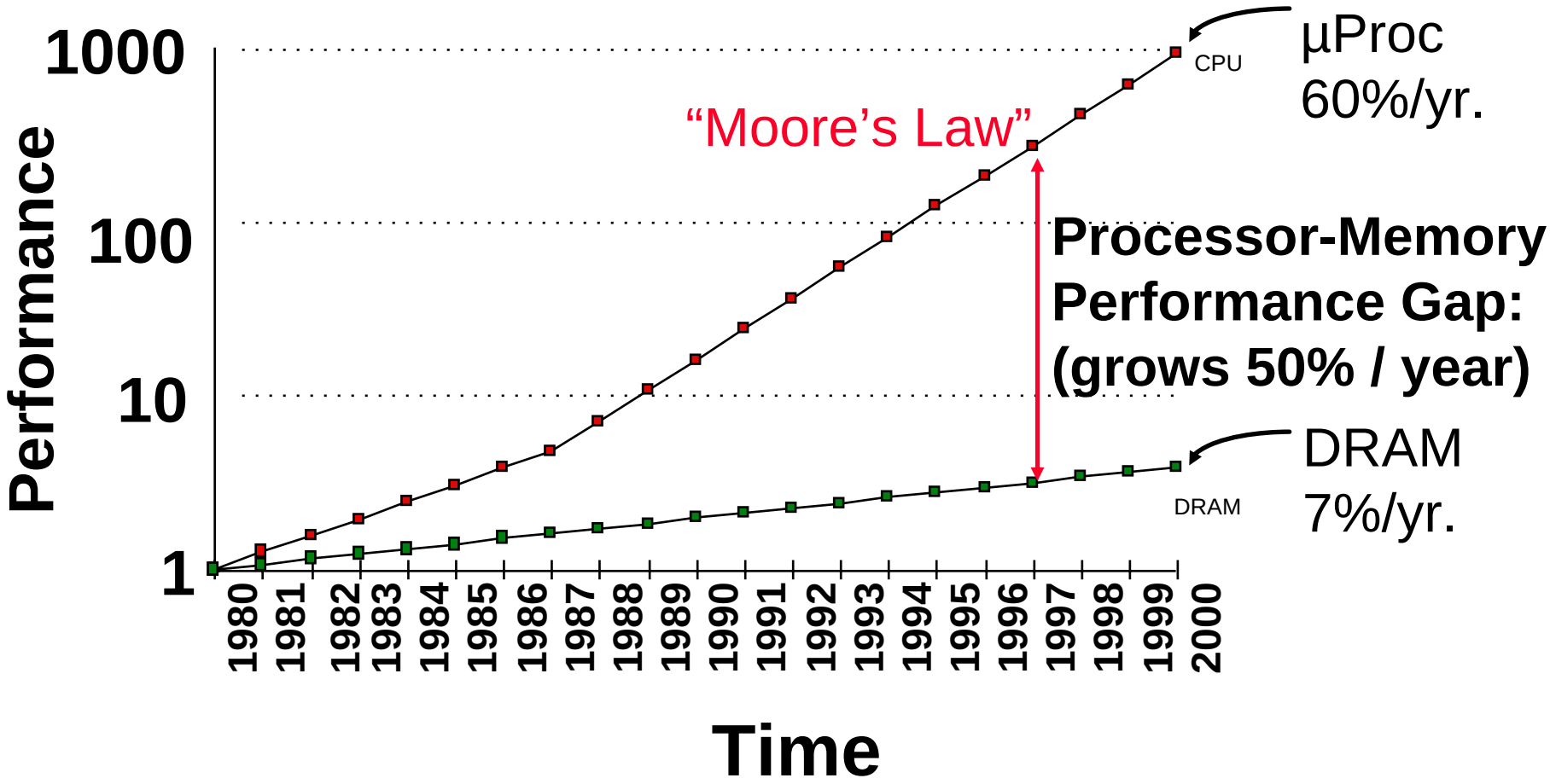
Uniprocessors in the Real World

- Real processors have
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs
  - parallelism
    - 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?

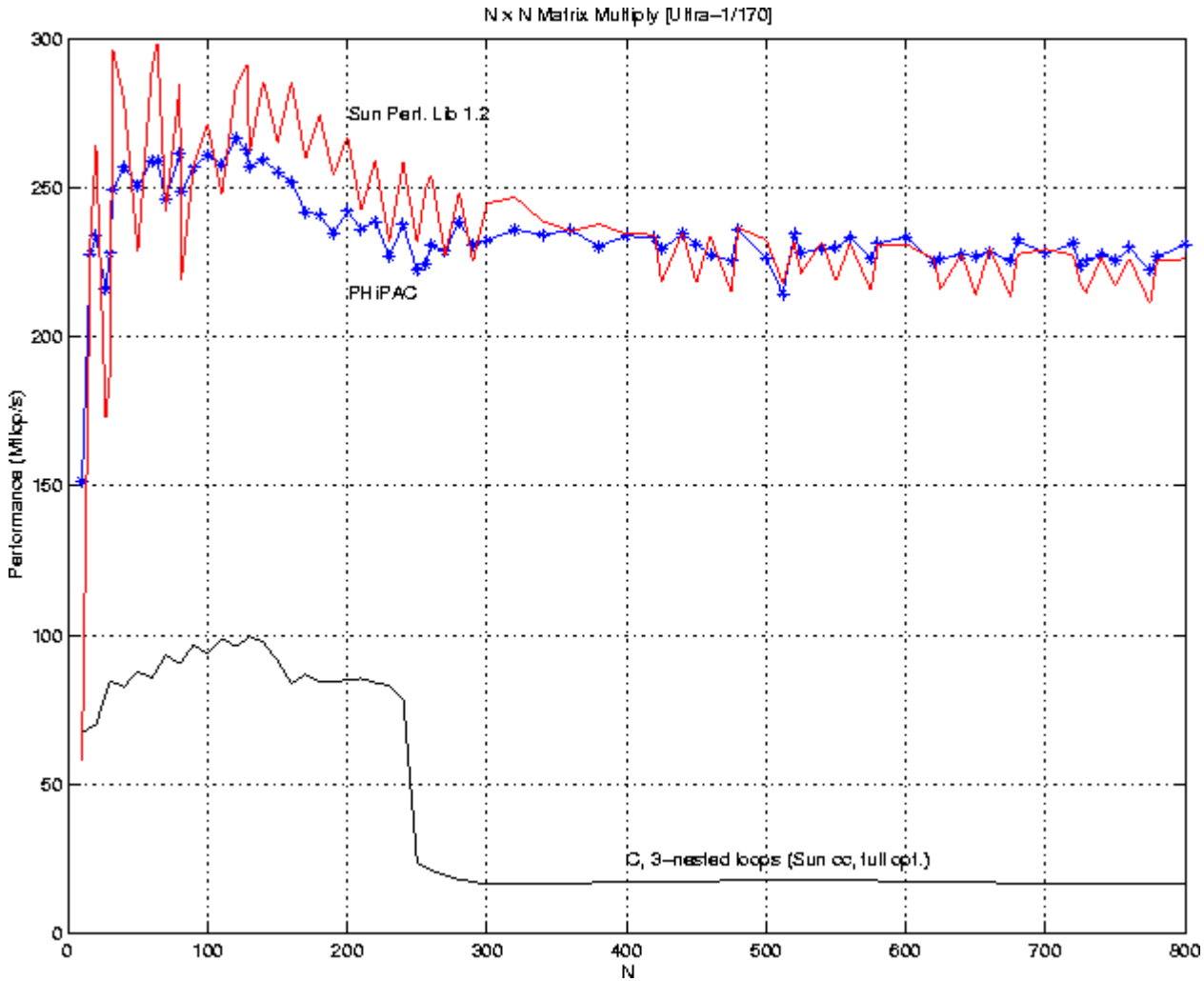
  In theory, compilers understand all of this and can optimize your program; in practice they don’t.

Processor-DRAM Gap (latency)

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



Matrix-multiply, optimized several ways



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

Cache and Its Importance in Performance

- Motivation:
  - Time to run code = clock cycles running code
    + clock cycles waiting for memory
  - For many years, CPU's have sped up an average of 50% per year over memory chip speed ups.
- Hence, memory access is the bottleneck to computing fast.
- Definition of a cache:
  - Dictionary: a safe place to hide or store things.
  - Computer: a level in a memory hierarchy.

Cache Sporting Terms

- Cache Hit: The CPU requests data that is already in the cache. We want to maximize this. The hit rate is the percentage of cache hits.
- Cache Miss: The CPU requests data that is not in cache. We want to minimize this. The miss time is how long it takes to get data, which can be variable and is highly architecture dependent.
- Two level caches are common. The L1 cache is on the CPU chip and the L2 cache is separate. The L1 misses are handled faster than the L2 misses in most designs.
- Upstream caches are closer to the CPU than downstream caches. A typical Alpha CPU has L1-L3 caches. Some MIPS CPU's do, too.

Cache Benefits

- Data cache was designed with two key concepts in mind
  - Spatial Locality
    - When an element is referenced its neighbors will be referenced too
    - Cache lines are fetched together
    - Work on consecutive data elements in the same cache line
  - Temporal Locality
    - When an element is referenced, it might be referenced again soon
    - Arrange code so that data in cache is reused often

Lessons

- 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
  - To write fast programs, need to consider architecture
  - We would like simple models to help us design efficient algorithms
  - Is this possible?
- We will illustrate with a common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

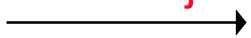
Note on Matrix Storage

- 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

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

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 element access

Key to algorithm efficiency

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

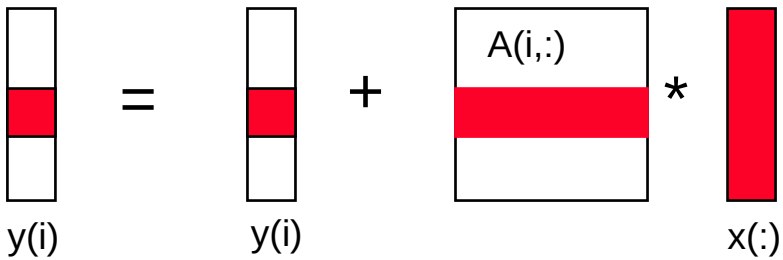
Key to machine efficiency

- Larger q means Time closer to minimum $f * t_f$

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

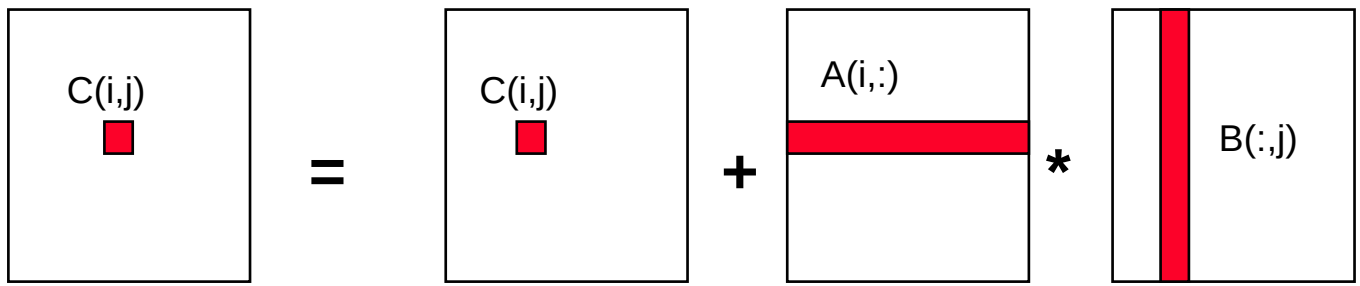
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
      {read row i of A into fast memory}
      for j = 1:n
            y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- m = number of slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- q = f / m ~= 2

- Matrix-vector multiplication limited by slow memory speed

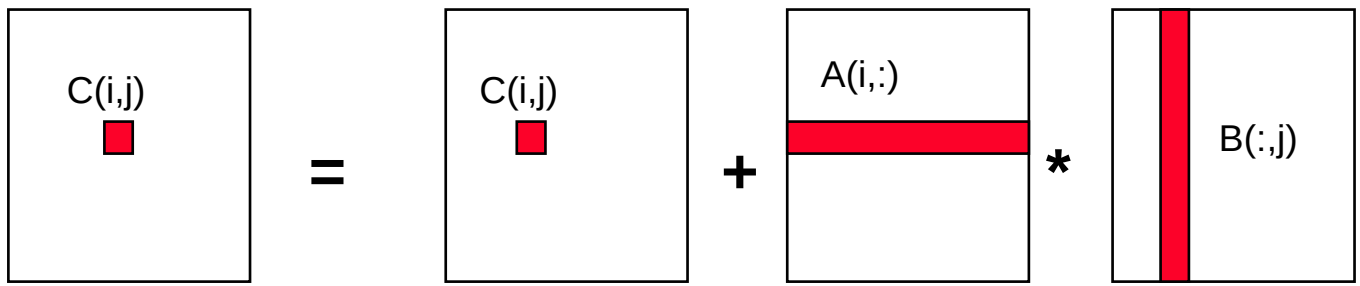
“Naïve” Matrix Multiply

```
{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

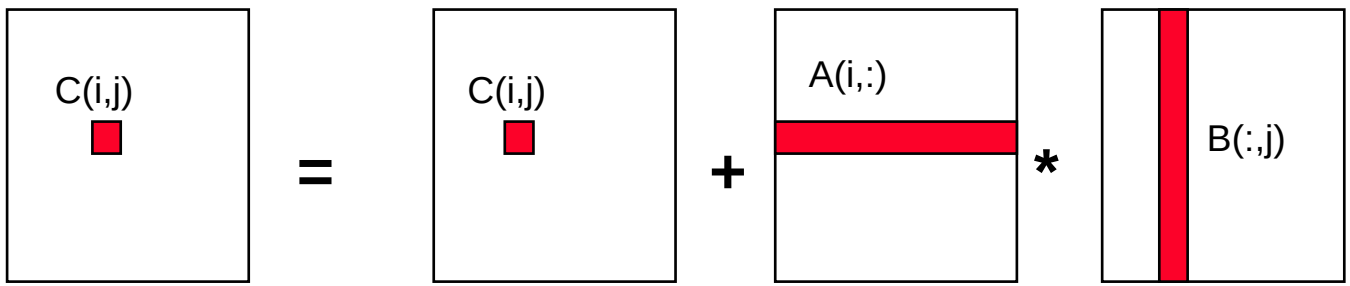
$+ n^2$ read each row of A once

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

$= n^3 + 3n^2$

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

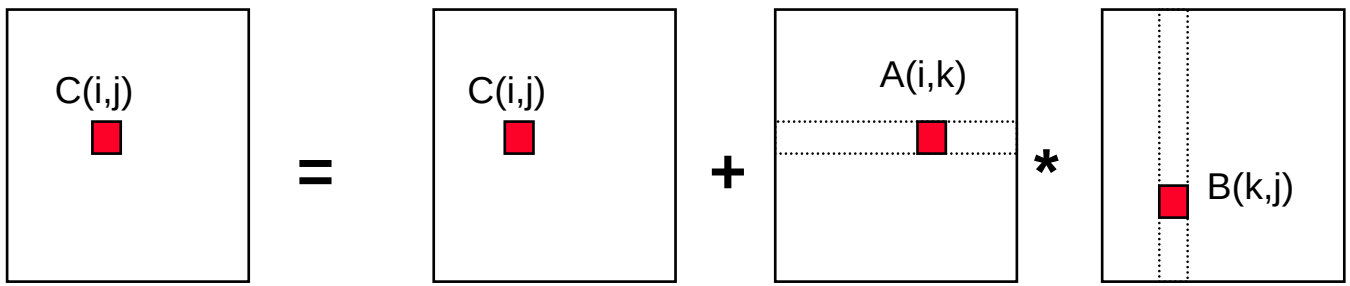
$\sim= 2$ for large n, no improvement over matrix-vector multiply



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 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 algorithm efficiency in the memory system

The amount of memory traffic is

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

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

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

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

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

Hence we can improve performance by increasing the blocksize b

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

Limits to Optimizing Matrix Multiply

The blocked algorithm has ratio q ~= b

- The large 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:

$3b^2 <= M$, so q ~= b <= sqrt(M/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))

Fast linear algebra kernels: BLAS

- Simple linear algebra kernels such as matrix-matrix multiply
- More complicated algorithms can be built from these basic kernels.
- The interfaces of these kernels have been standardized as the Basic Linear Algebra Subroutines (BLAS).
- Early agreement on standard interface (~1980)
- Led to portable libraries for vector and shared memory parallel machines.
- On distributed memory, there is a less-standard interface called the PBLAS

BLAS: advantages

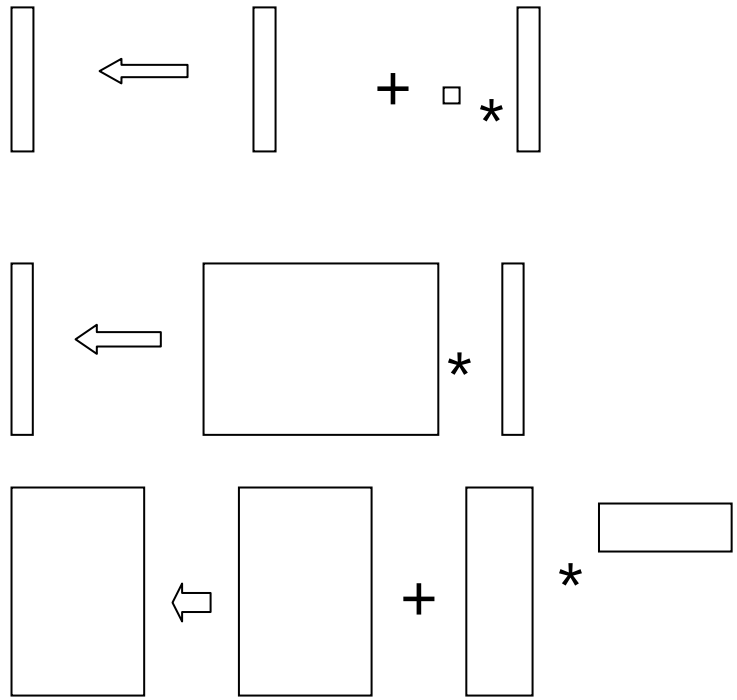
- Clarity: code is shorter and easier to read,
- Modularity: gives programmer larger building blocks,
- Performance: manufacturers will provide tuned machine-specific BLAS,
- Program portability: machine dependencies are confined to the BLAS

Basic Linear Algebra Subroutines

- 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: 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: matrix matrix multiply, etc
    - m >= 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 (e.g., LAPACK)
- See `www.netlib.org/blas, www.netlib.org/lapack`

Level 1, 2 and 3 BLAS

- Level 1 BLAS Vector-Vector operations
- Level 2 BLAS Matrix-Vector operations
- Level 3 BLAS Matrix-Matrix operations



Level 1 BLAS

- Operate on vectors or pairs of vectors
  - perform O(n) operations;
  - return either a vector or a scalar.
- saxpy
  - y(i) = a * x(i) + y(i), for i=1 to n.
  - s stands for single precision, daxpy is for double precision, caxpy for complex, and zaxpy for double complex,
- sscal y = a * x, for scalar a and vectors x,y
- sdot computes $s = S^{n}_{i=1}\ x(i)*y(i)$

Level 2 BLAS

- Operate on a matrix and a vector;
  - return a matrix or a vector;
  - $O(n^2)$ operations
- *sgemv*: matrix-vector multiply
  - y = y + A*x
  - where A is m-by-n, x is n-by-1 and y is m-by-1.
- *sger:* rank-one update
  - $A = A + y*x^T$, i.e., A(i,j) = A(i,j)+y(i)*x(j)
  - where A is m-by-n, y is m-by-1, x is n-by-1,
  - strsv: triangular solve
  - solves y=T*x for x, where T is triangular

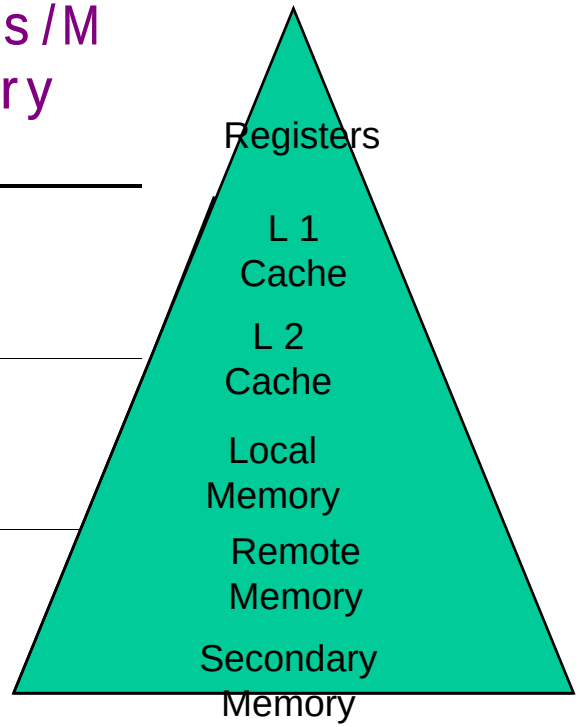
Level 3 BLAS

- Operate on pairs or triples of matrices
  - returning a matrix;
  - complexity is $O(n^3)$.
- sgemm: Matrix-matrix multiplication
  - C = C +A*B,
  - where C is m-by-n, A is m-by-k, and B is k-by-n
- strsm: multiple triangular solve
  - solves Y = T*X for X,
  - where T is a triangular matrix, and X is a rectangular matrix.

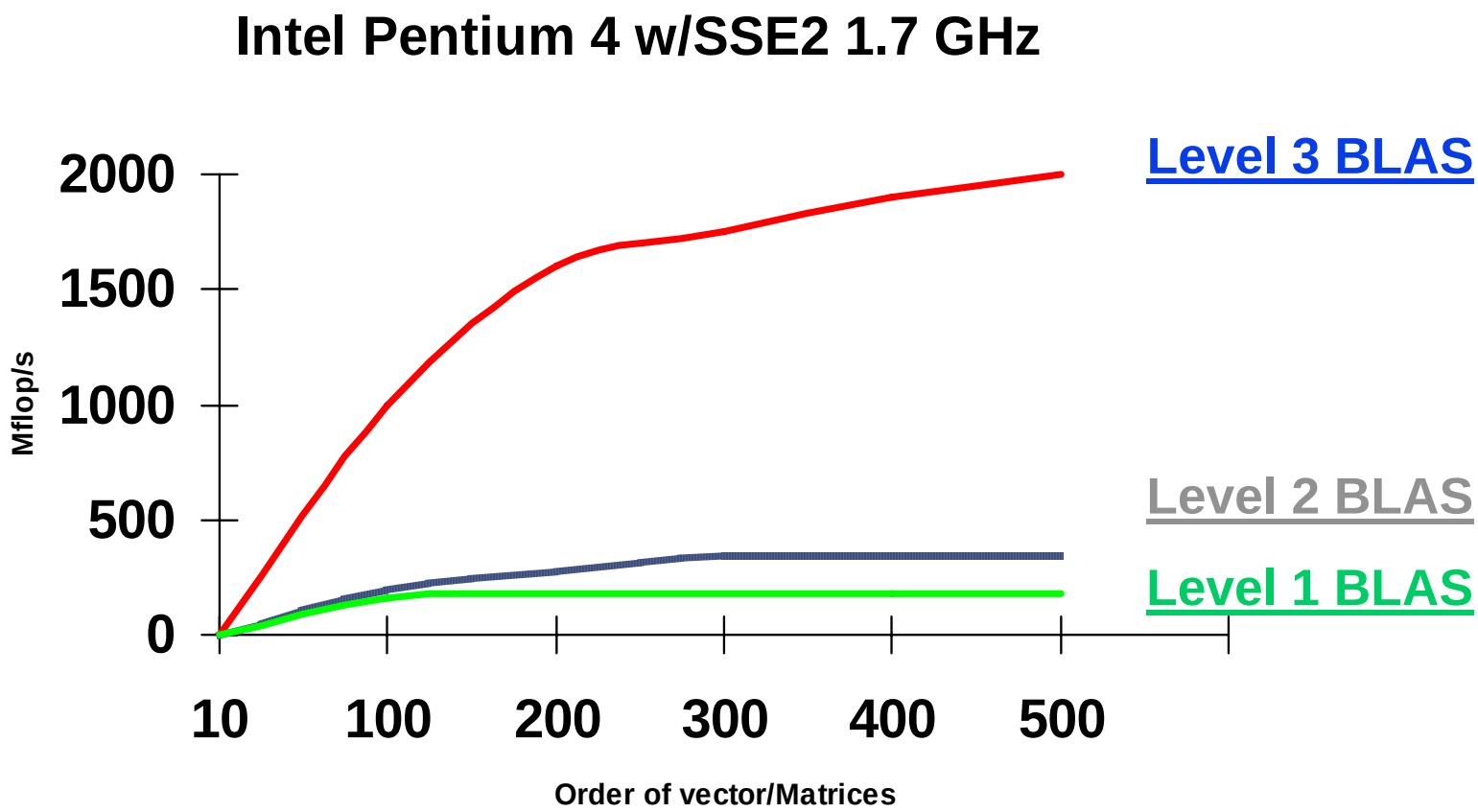
Why Higher Level BLAS?

- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

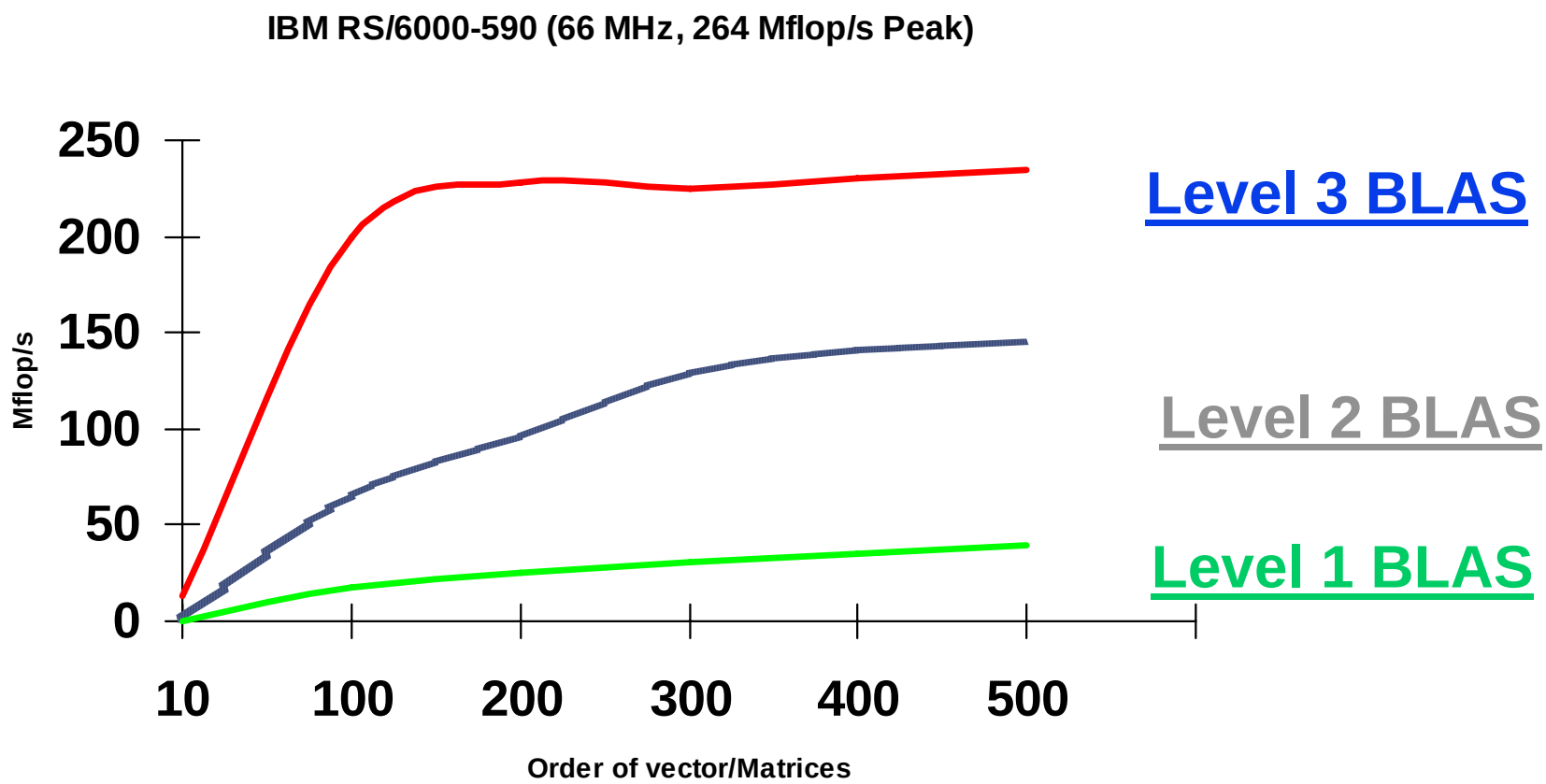
BLAS	Memory Refs	Flops	Flops/Memory Refs
Level 1 $y = y + \alpha x$	$3n$	$2n$	$2/3$
Level 2 $y = y + A x$	n^2	$2n^2$	2
Level 3 $C = C + A B$	$4n^2$	$2n^3$	$n/2$



BLAS for Performance



BLAS for Performance

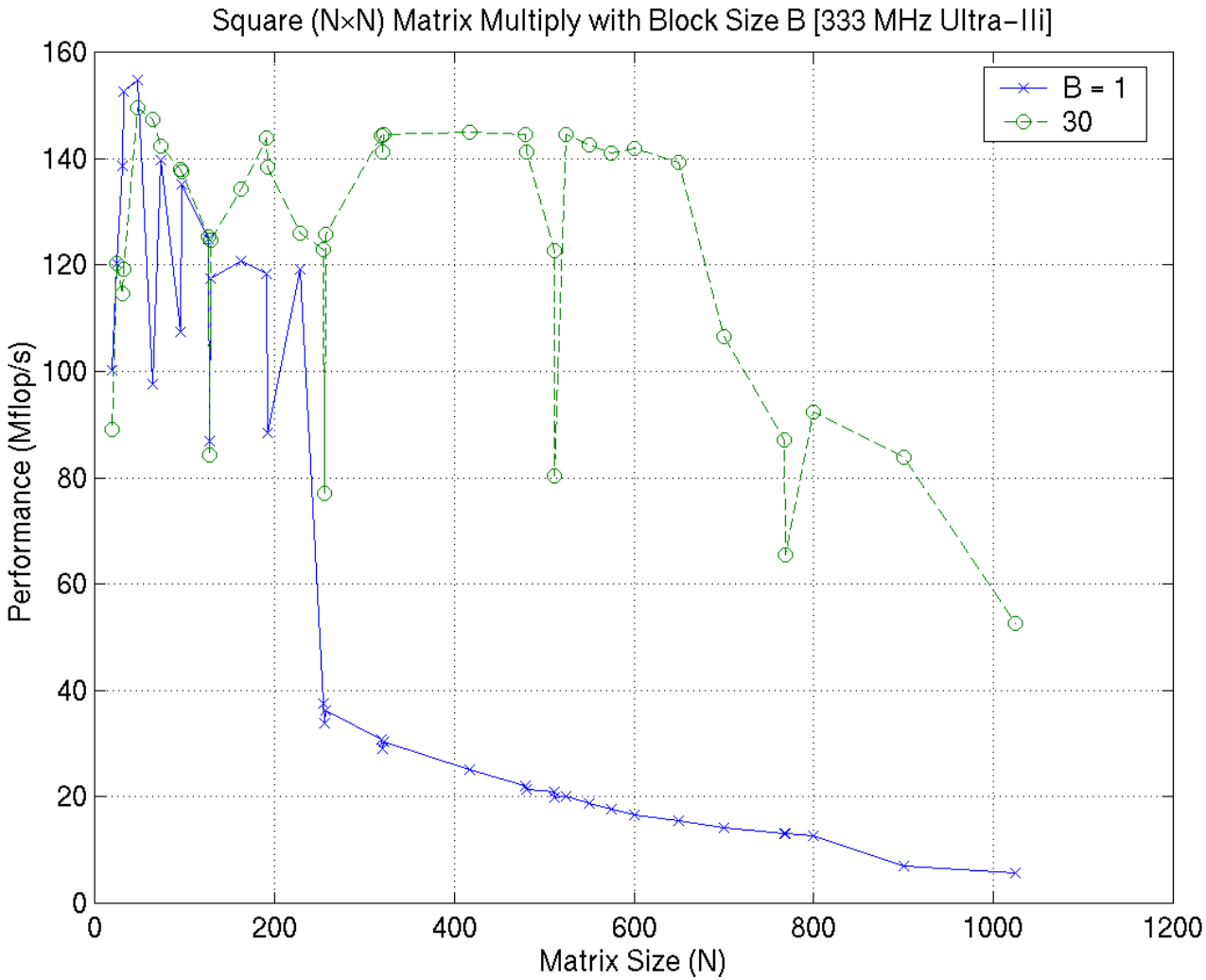


Locality in Other Algorithms

- 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)

Tiling (Blocking) Alone Might Not Be Enough

- Naïve and a “naïvely tiled” code

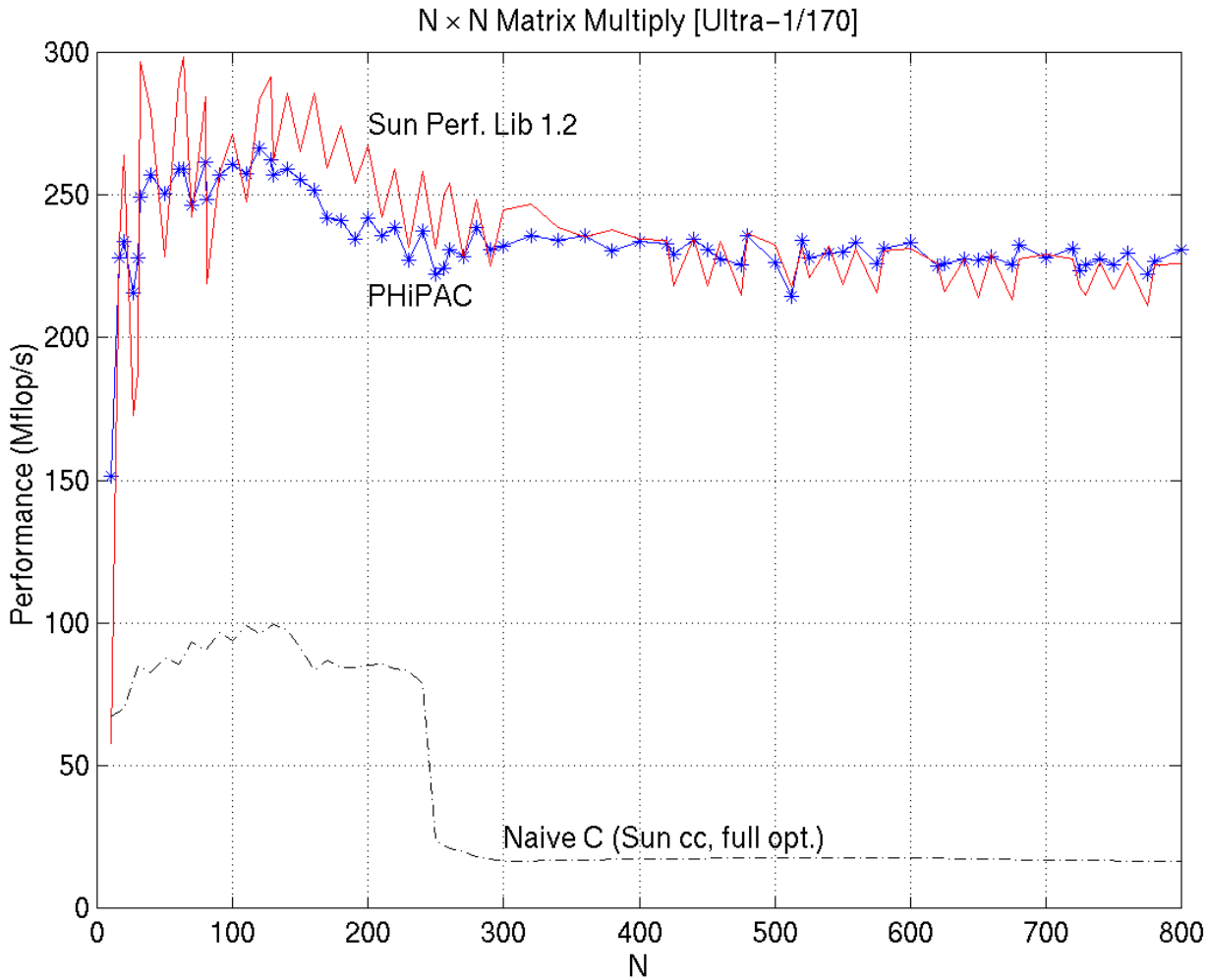


Optimizing in Practice

- 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
- Automatic optimization an active research area
  - BeBOP: `www.cs.berkeley.edu/~richie/bebop`
  - PHiPAC: `www.icsi.berkeley.edu/~bilmes/phipac`

    in particular `tr-98-035.ps.gz`
  - ATLAS: `www.netlib.org/atlas`

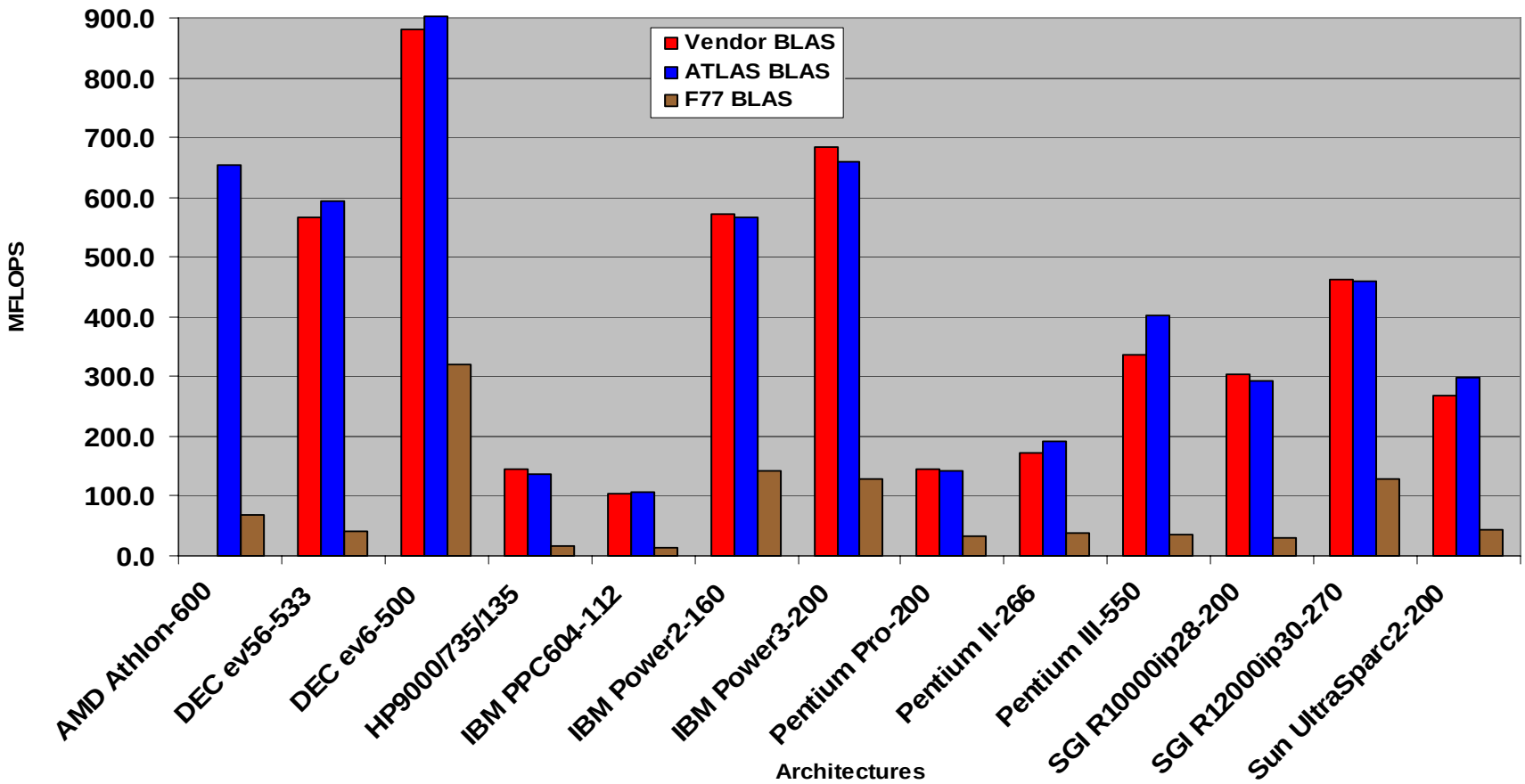
PHiPAC: Portable High Performance ANSI C



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. (being incorporated in MATLAB)

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

Supplement: Strassen’s aglorithm

Conventional Block Matrix Multiply

2 by 2 block matrix multiply:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Strassen's algorithm

Strassen does it with 7 multiplies (but many more adds)

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22})B_{11}$$
$$P_3 = A_{11}(B_{12} - B_{22})$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12})B_{22}$$
$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_5$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

One matrix multiply is replaced by 14 matrix additions

Strassen's algorithm

- The count of arithmetic operations is:

	Mult	Add	Complexity
Regular	8	4	$2n^3+O(n^2)$
Strassen	7	18	$4.7n^{2.8} + O(n^2)$

- Current world's record is O(n^2.376…)
- In reality the use of Strassen's algorithm is limited by
  - Additional memory required for storing the P matrices.
  - More memory accesses are needed.