Memory Hierarchies and Optimizations: Case Study in Matrix Multiplication

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07

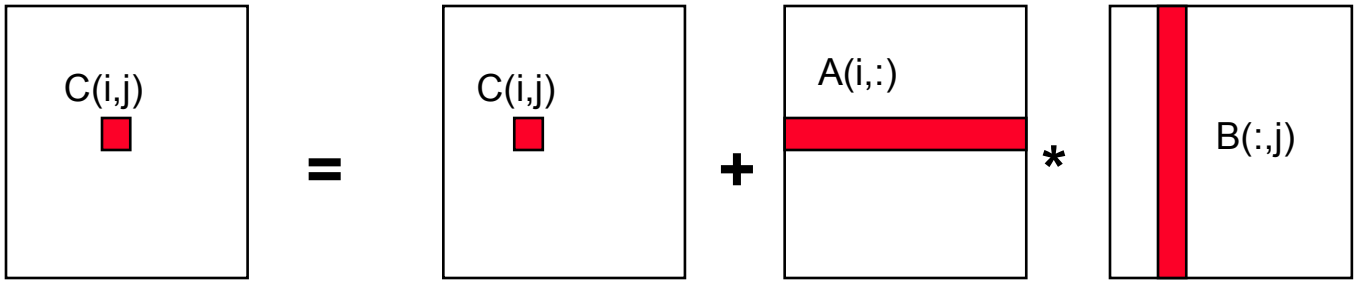
Naïve Matrix Multiply

```
{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

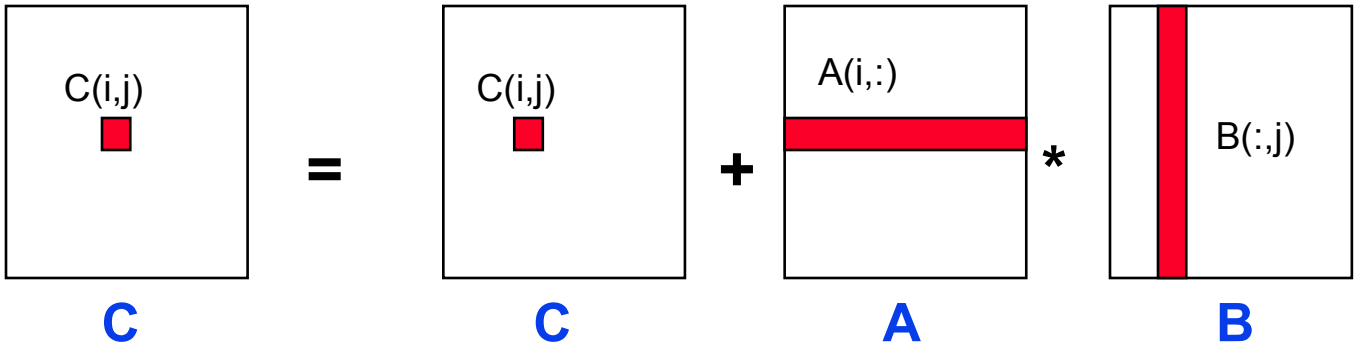
Reuse quotient (q = flops/word) in the algorithm is *potentially* as large as

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



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

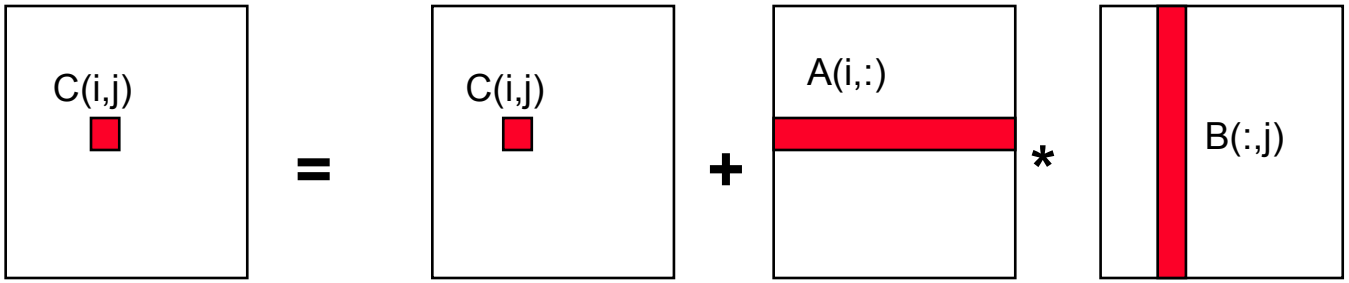
  $+ 2n^2$ to read and write each element of C once

  $= n^3 + 3n^2$

- So the re-use quotient, $q = f / m = 2n^3 / (n^3 + 3n^2)$

  $\sim= 2$ for large n

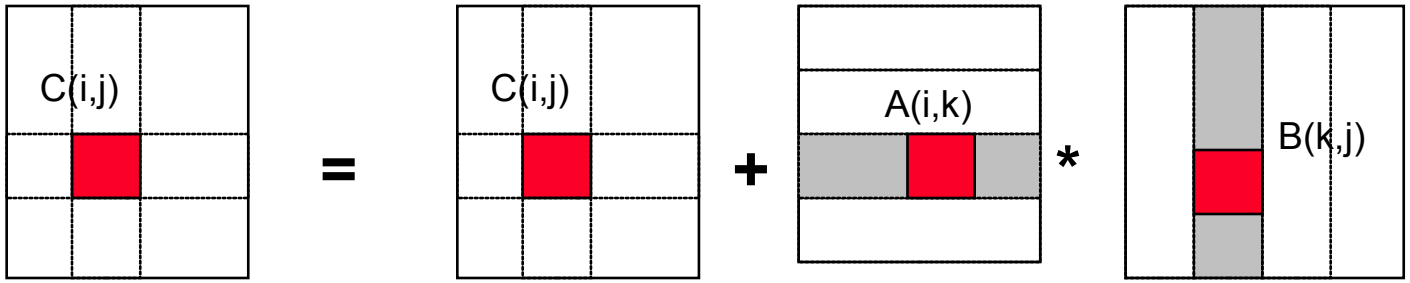
- This is no better than matrix-vector multiply
- And is far from the "best possible" which is 2/3*n for large n
- And this doesn't take into account cache lines



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, also NxN blocks each of size bxb (N=n/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

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

- Re-use quotient is: $q = f / m = 2n^3 / ((2N + 2) * n^2)$
  $\sim= n / N = b$ for large n
- 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 ~= 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}

$$3b^2 <= M_{fast}, \text{ so } q \sim= b <= \text{sqrt}(M_{fast}/3)$$

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

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

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

		required
	t_m/t_f	**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

Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each C[i,j] by applying associativity
  - Get slightly different answers from naïve code, because of roundoff - OK
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \sim= b <= \text{sqrt}(M_{fast}/3)$$

- Aside (for those who have taken CS170 or equivalent)
- There is a *lower bound* result that says we cannot do any better than this (using only associativity)

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

- What if more levels of memory hierarchy?

Tiling for Multiple Levels

- Multiple levels: pages, caches, registers in machines
- Each level could have it's only 3-nested loops:

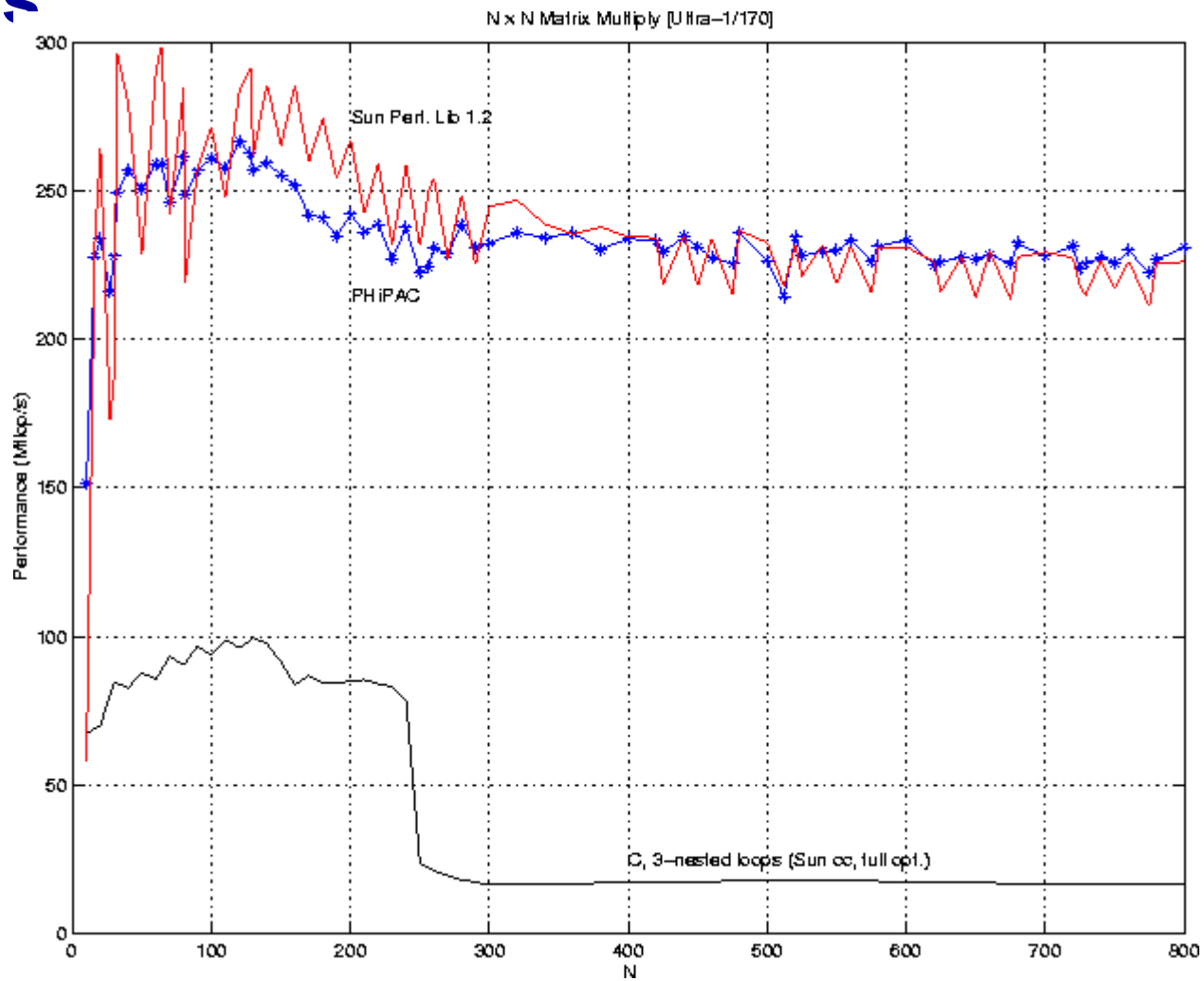
```
for i, for j, for k  {matul blocks that fit in L2 cache}
    for ii, for jj, for kk {matmul blocks that fit in L1}
        for iii, for jjj, for kkk {matmul blocks that fit in registers}
```

- But in practice don't use so many levels and fully unroll code for register "tiles"
  - E.g., if these are 2x2 matrix multiplies, can write out all 4 statements without loops

```
c[0,0] = c[0,0] + a [0,0] * b[0,0] + a[0,1]*b[1,0]
c[1,0] =
c[0,1] =
c[1,1] =
```

Many possible code variations; see Mark's notes on code generators in HW page

Matrix-multiply, optimized several ways



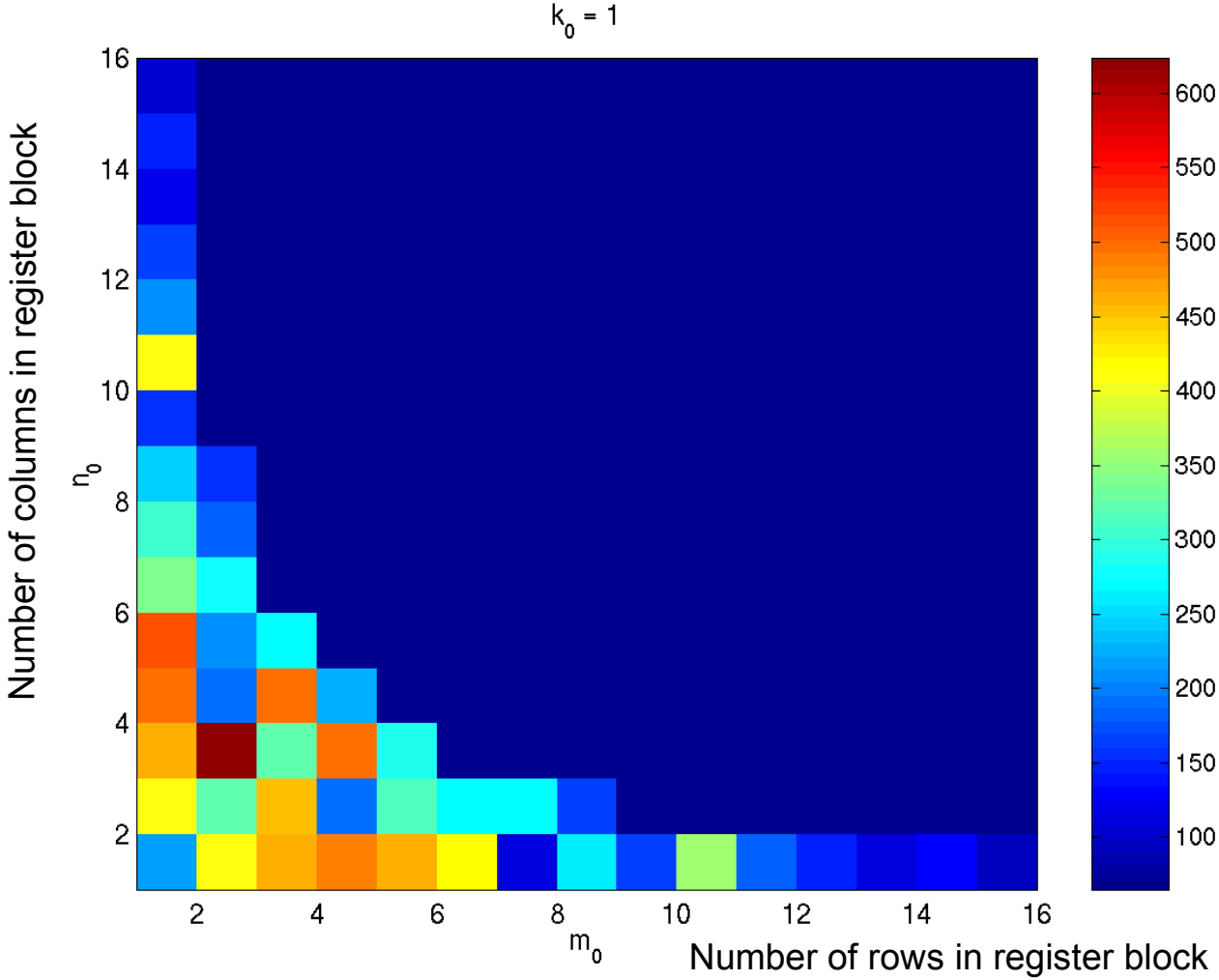
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

Search Over Block Sizes

Strategies for choose block sizes:

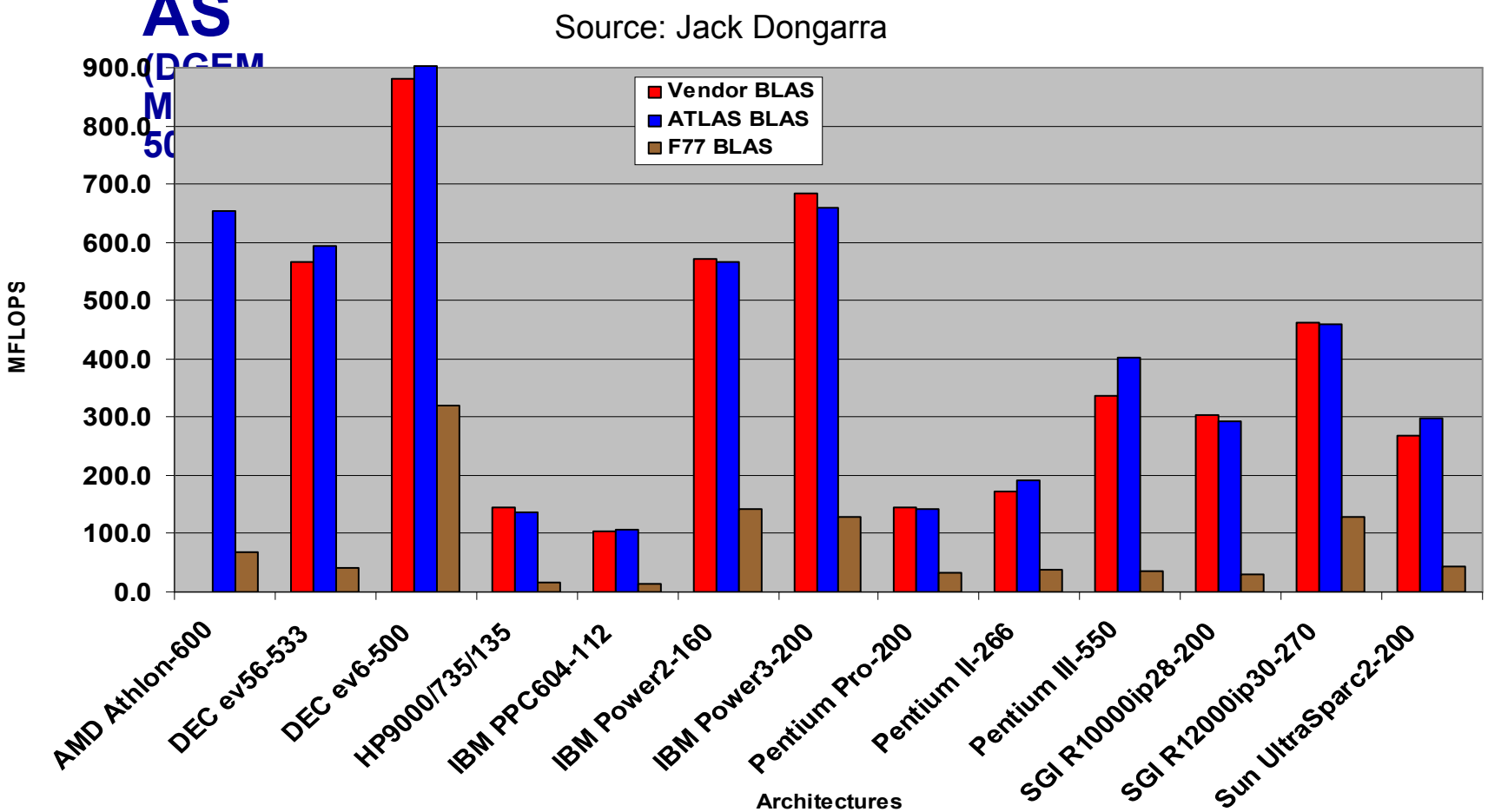
- Find out hardware parameters:
  - Read vendor manual for register #, cache sizes (not part of ISA), page sizes (OS config)
  - Measure yourself using memory benchmark from last time
  - But in practice these don't always work well; memory systems are complex as we saw
- Manually try many variations ☹
- Write "autotuner" to search over "design space" of possible implementations
  - Atlas – incorporated into Matlab
  - PhiPAC – original dense linear algebra tuning project from Berkeley; came from homework assignment like HW2

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-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

ATLAS (DGEMM, N=500)



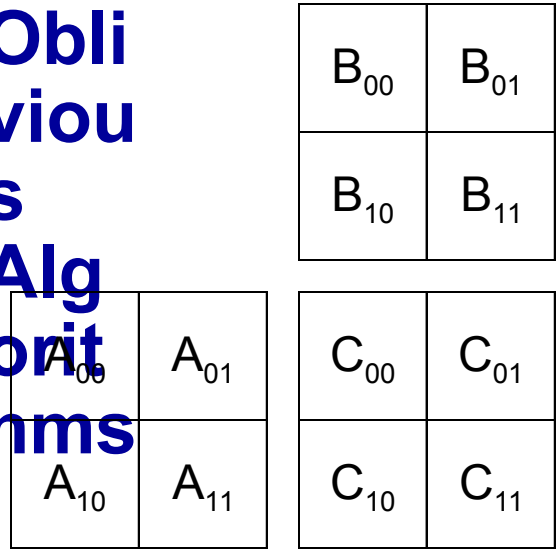
- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

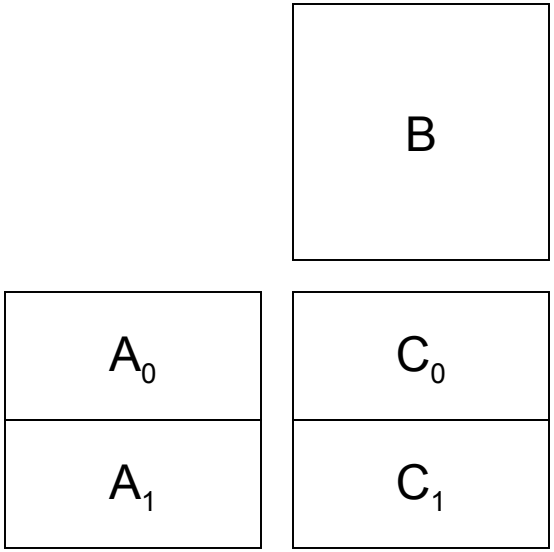
Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat nxn matrix multiply set of smaller problems
  - Eventually, these will fit in cache

- The idea of cache-oblivious algorithms is use for other problems than matrix multiply. The general idea is:
  - Think of recursive formulation
  - If the subproblems use smaller data sets and some reuse within that data set, then a recursive order may improve performance

Cache-Oblivious Algorithms





$$C_{00} = A_{00}*B_{00} + A_{01}*B_{10}$$
$$C_{01} = A_{01}*B_{11} + A_{00}*B_{01}$$
$$C_{11} = A_{11}*B_{01} + A_{10}*B_{01}$$
$$C_{10} = A_{10}*B_{00} + A_{11}*B_{10}$$

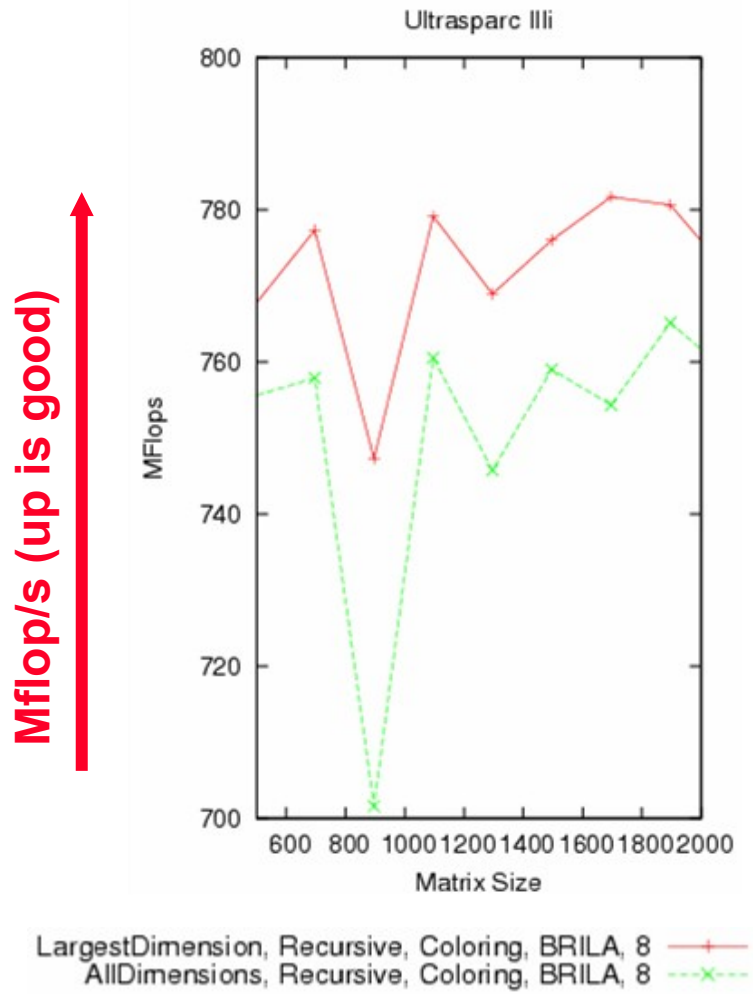
- Divide all dimensions (AD)
- 8-way recursive tree down to 1x1 blocks
  - Gray-code order promotes reuse
- Bilardi, et al.

$$C_0 = A_0*B$$
$$C_1 = A_1*B$$

- Divide largest dimension (LD)
- Two-way recursive tree down to 1x1 blocks
- Frigo, Leiserson, et al.

Cache-Oblivious: Discussion

- Block sizes
  - Generated dynamically at each level in the recursive call tree
- Our experience
  - Performance is similar
  - Use AD for the rest of the talk

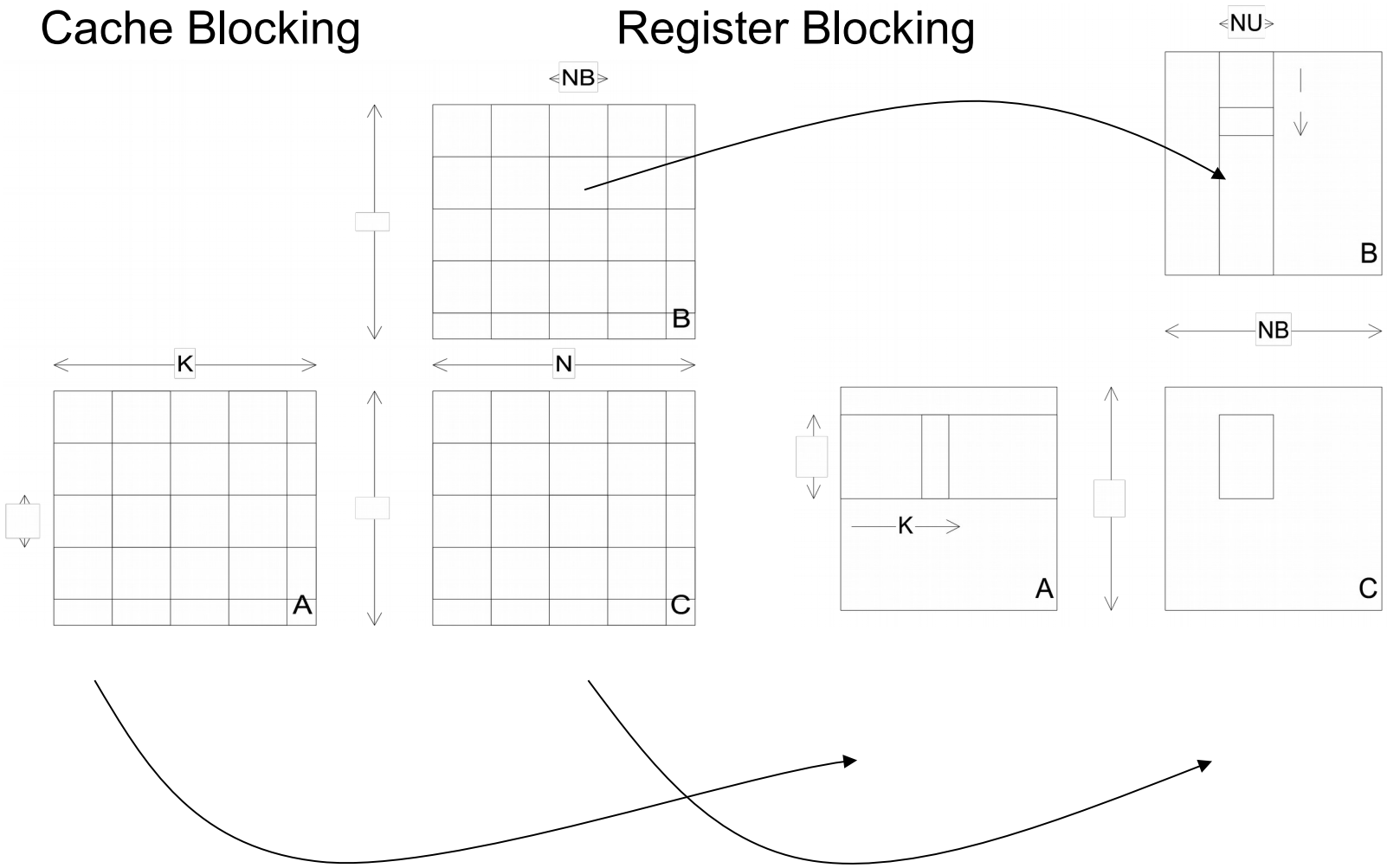


 Slide source: Roeder, Yotov, Pingali at Cornell/UTexas

Cache-Conscious Algorithms

- Usually Iterative
  - Nested loops
- Implementation of blocking
  - Cache blocking achieved by Loop Tiling
  - Register blocking also requires Loop Unrolling

Structure of Tiled (Cache Concious) Code

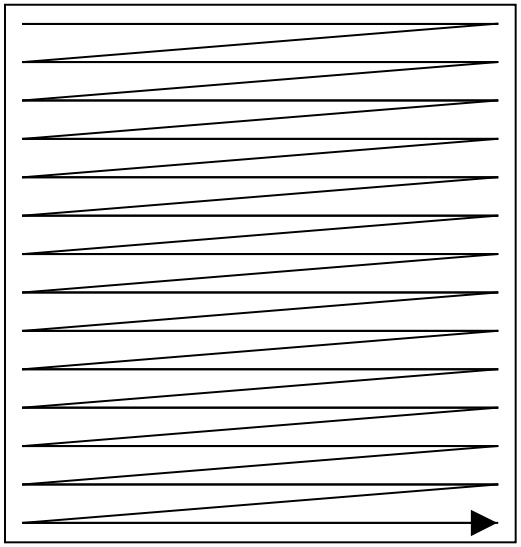


 Slide source: Roeder, Yotov, Pingali at Cornell/UTexas

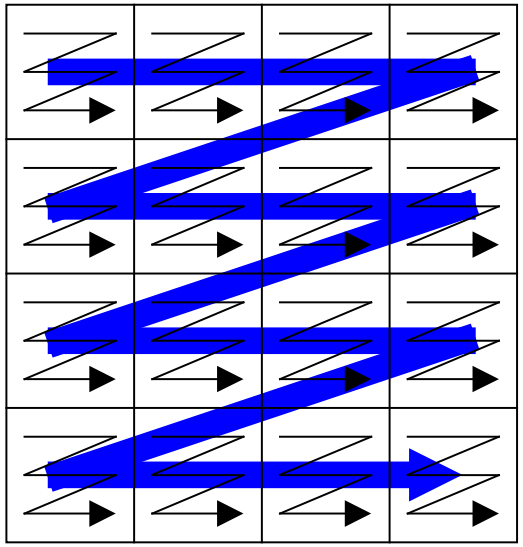
Data Structures

- Fit control structure better
- Improve
  - Spatial locality
  - Streaming, prefecthing

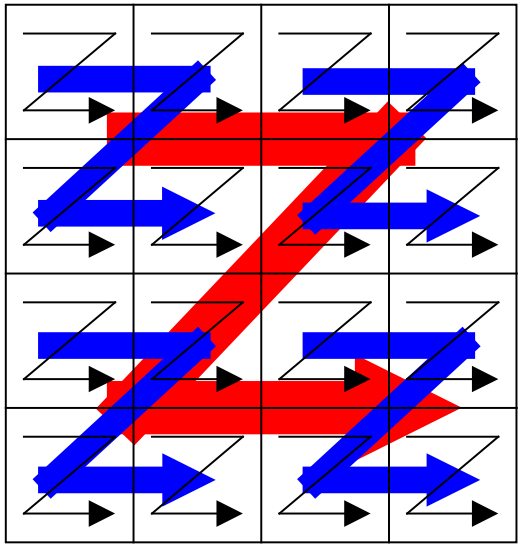
Row-major



Row-Block-Row

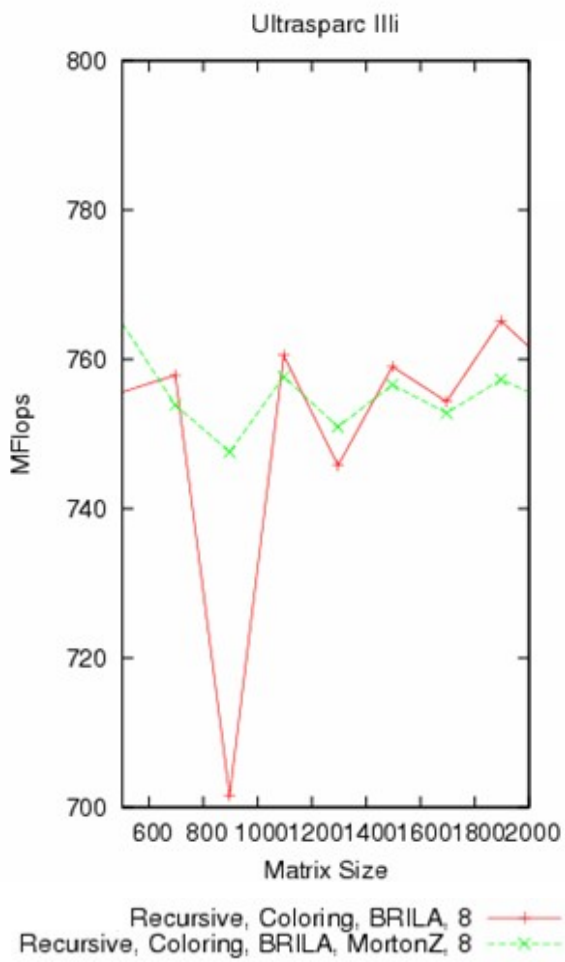


Morton-Z



Data Structures: Discussion

- Morton-Z
  - Matches recursive control structure better than RBR
  - Suggests better performance for CO
  - More complicated to implement
  - In our experience payoff is small or even negative
- Use RBR for the rest of the talk

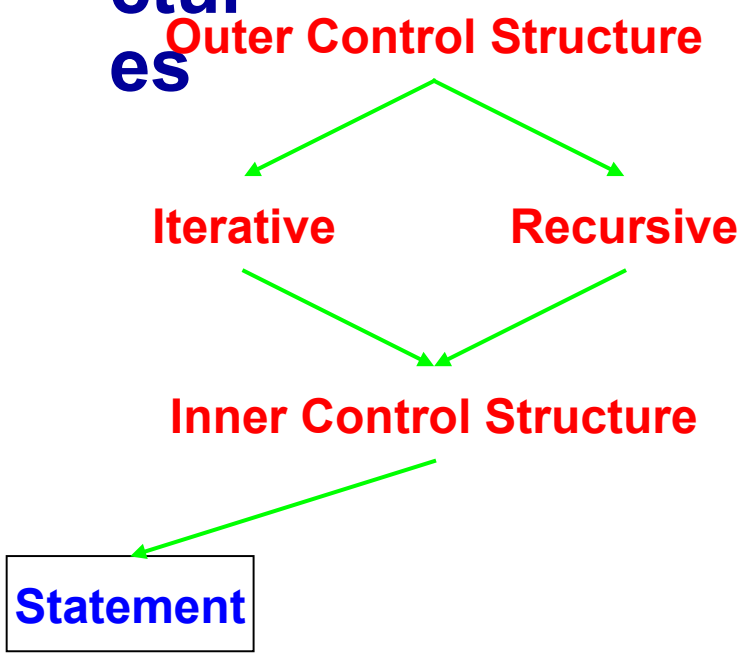


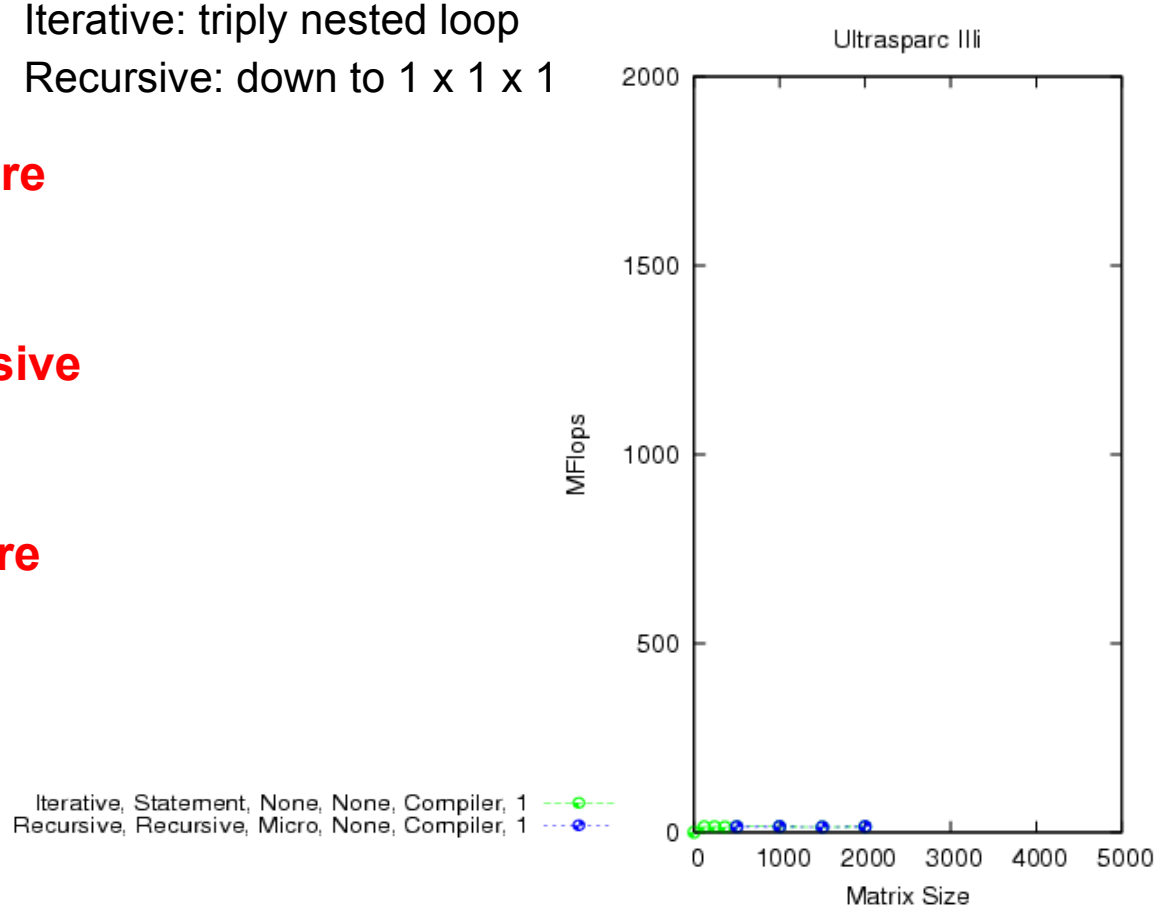
UltraSPARC III

- Peak performance
  - 2 GFlops
- Memory hierarchy
  - Registers: 32
  - L1 data cache: 64KB, 4-way
  - L2 data cache: 1MB, 4-way
- Compilers
  - FORTRAN: SUN F95 7.1
  - C: SUN C 5.5

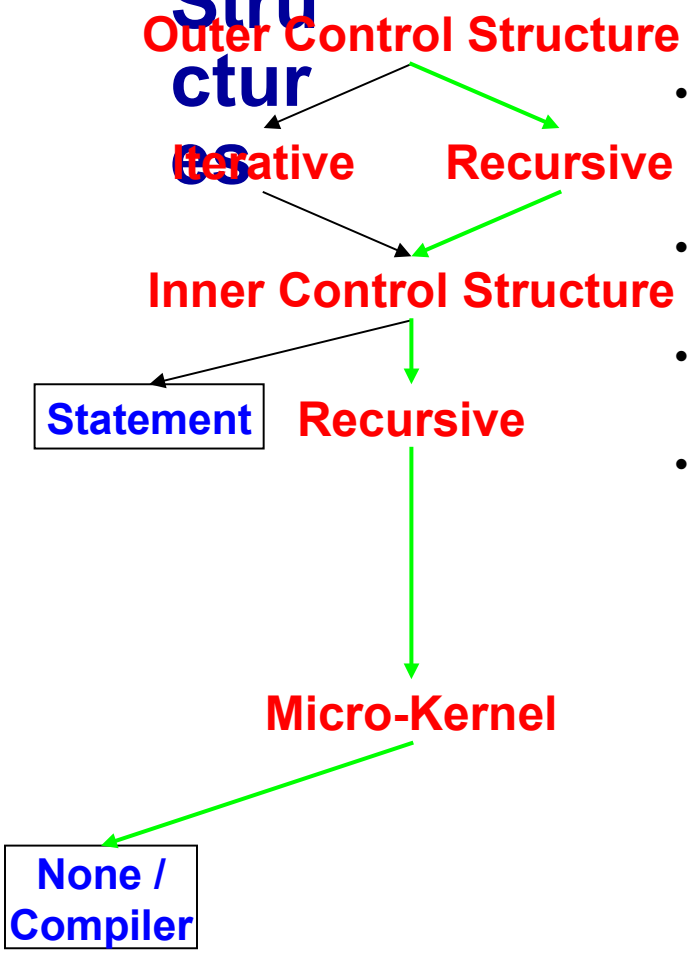
Control Structures

- Iterative: triply nested loop
- Recursive: down to 1 x 1 x 1

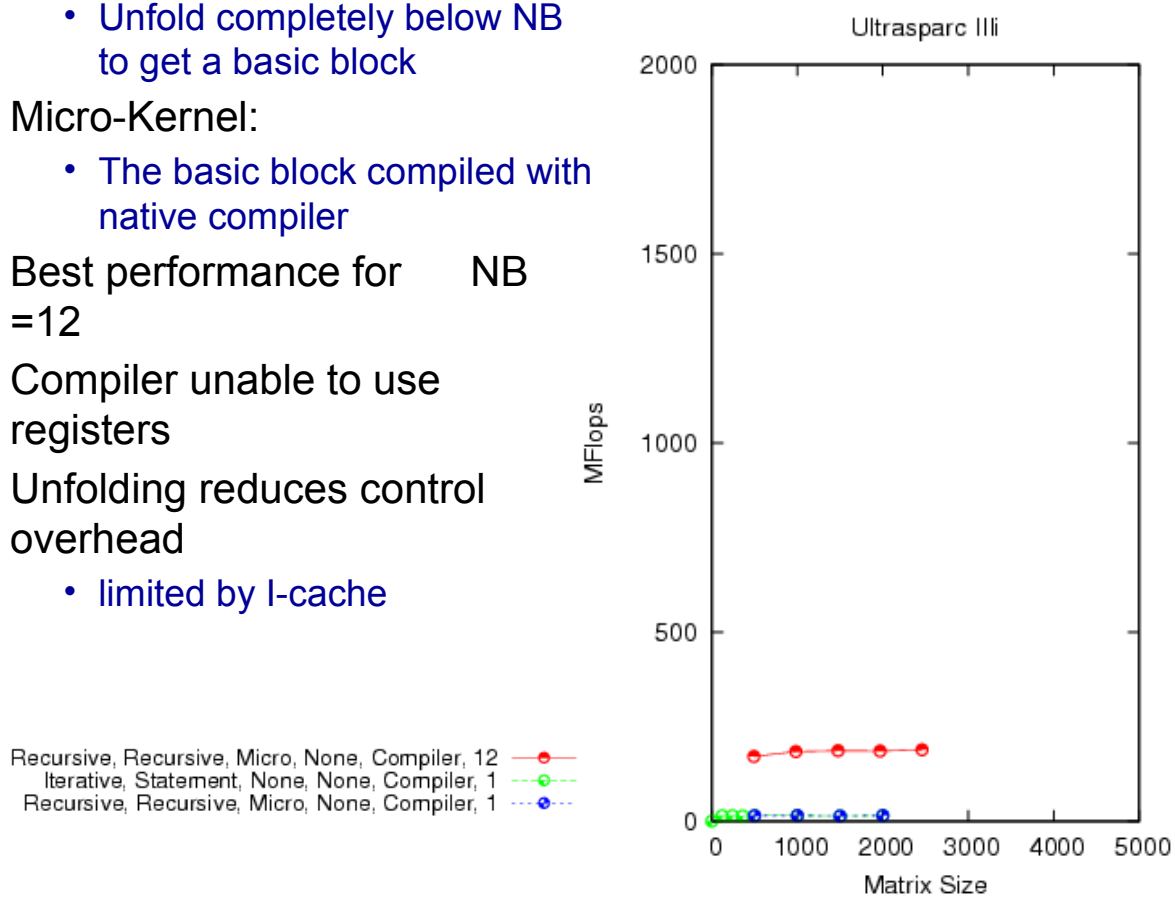




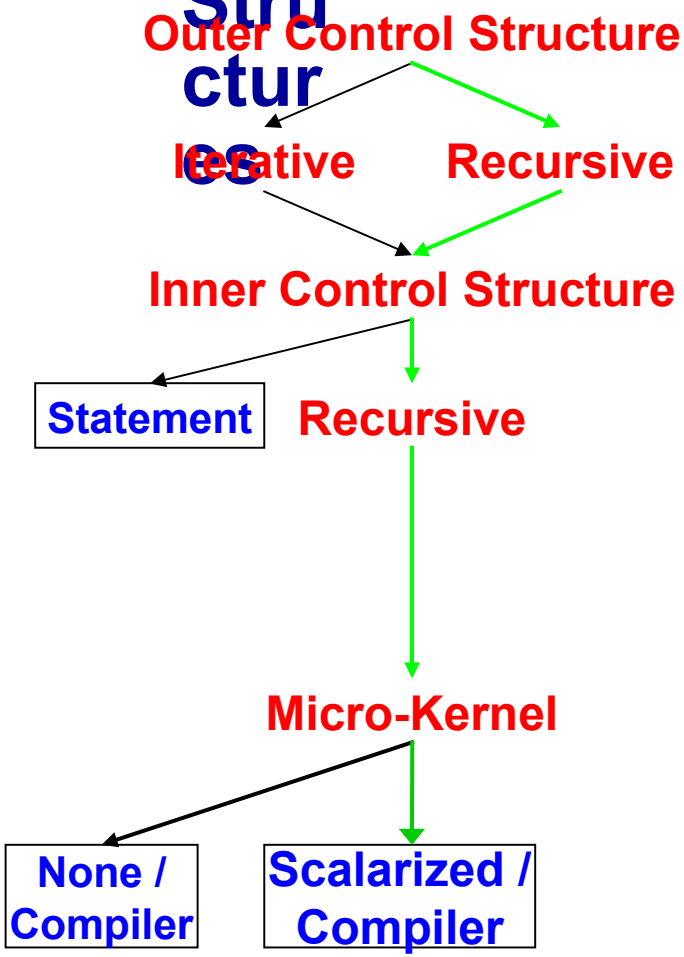
Control Structures



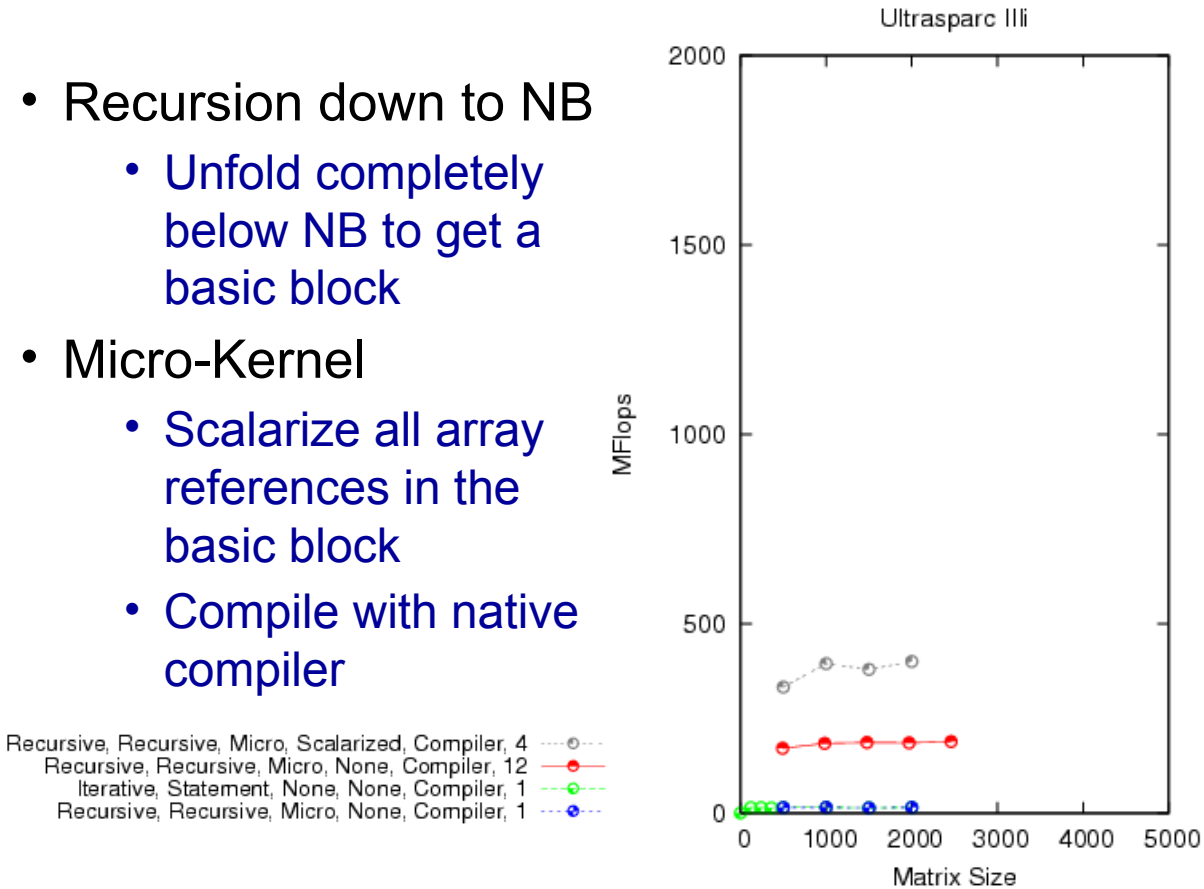
- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel:
  - The basic block compiled with native compiler
- Best performance for NB =12
- Compiler unable to use registers
- Unfolding reduces control overhead
  - limited by I-cache



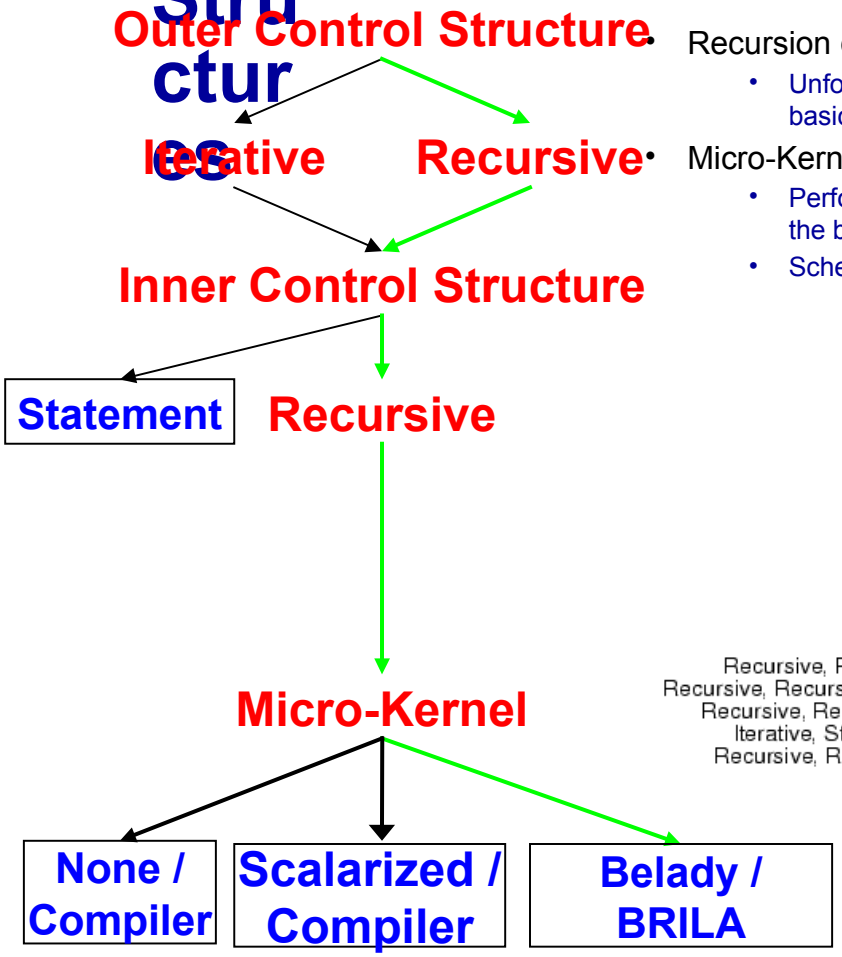
Control Structures



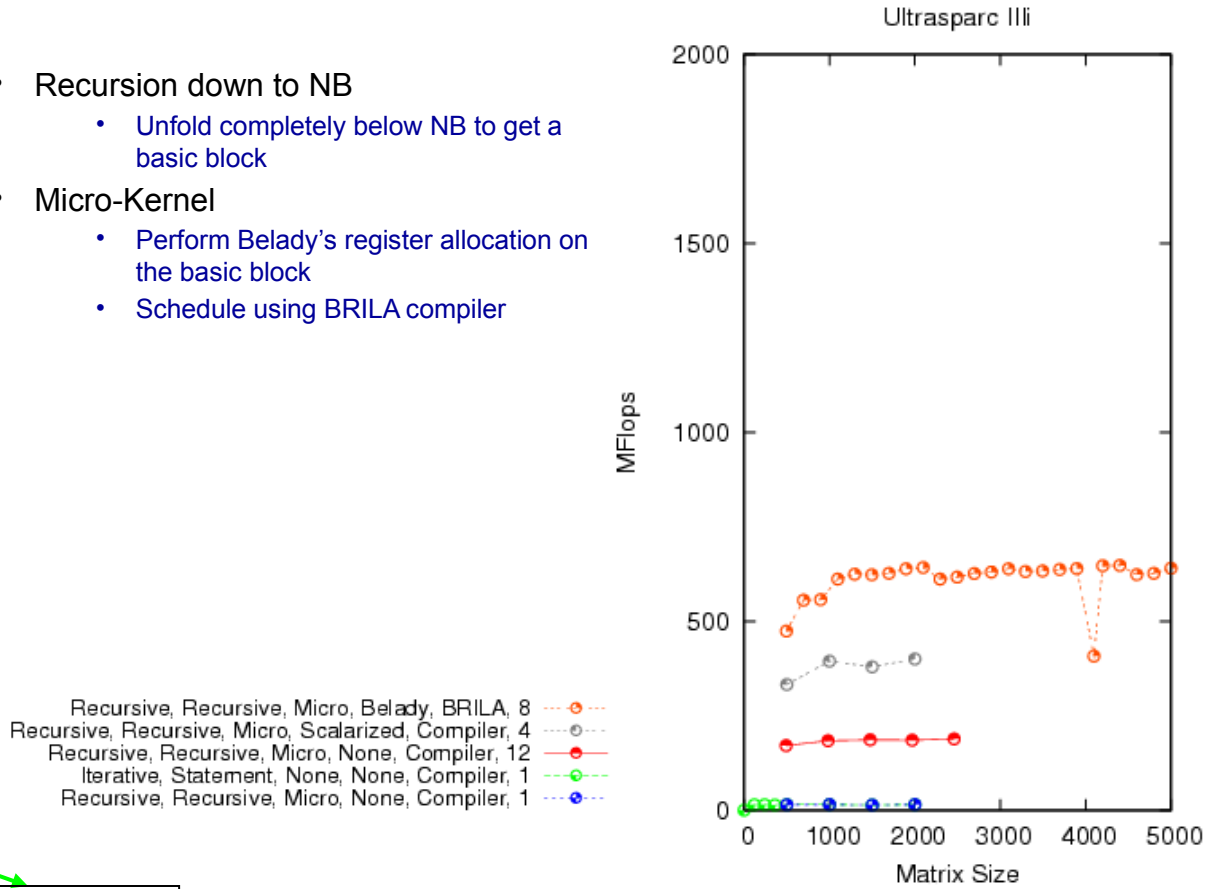
- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Scalarize all array references in the basic block
  - Compile with native compiler



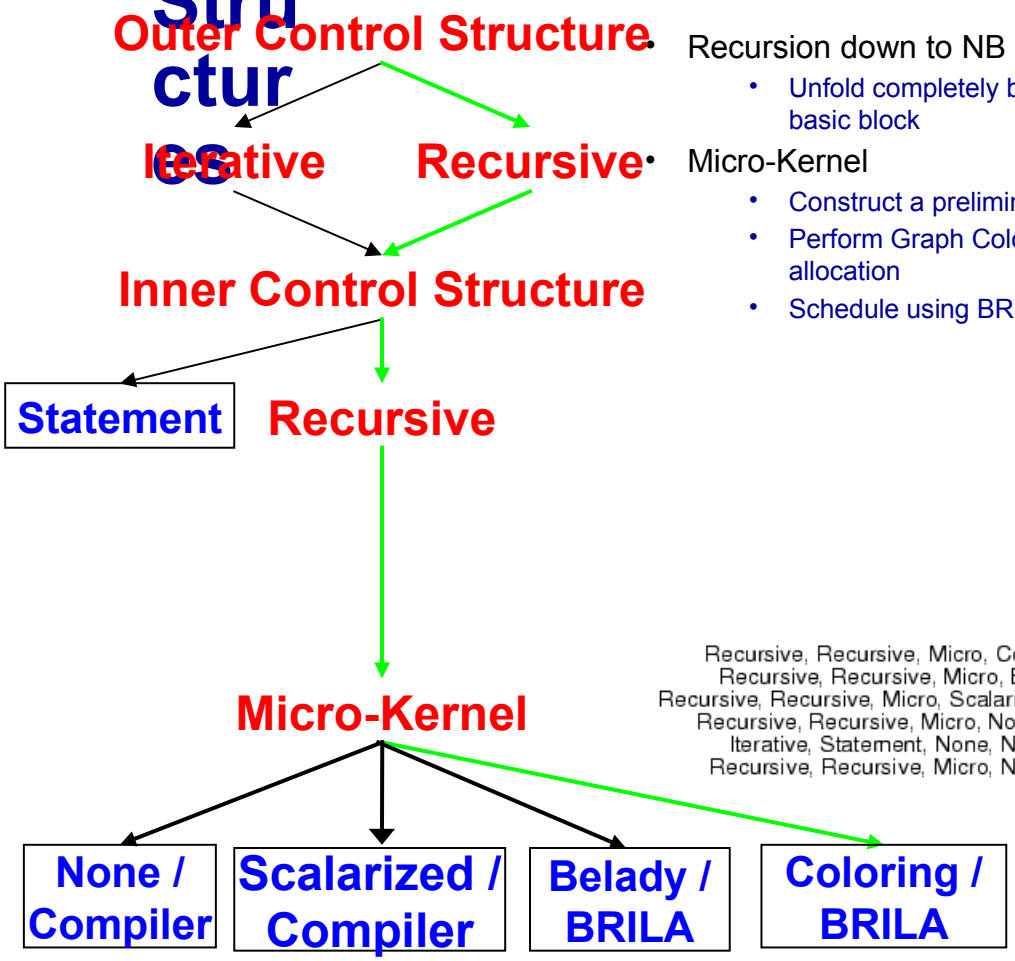
Control Structures



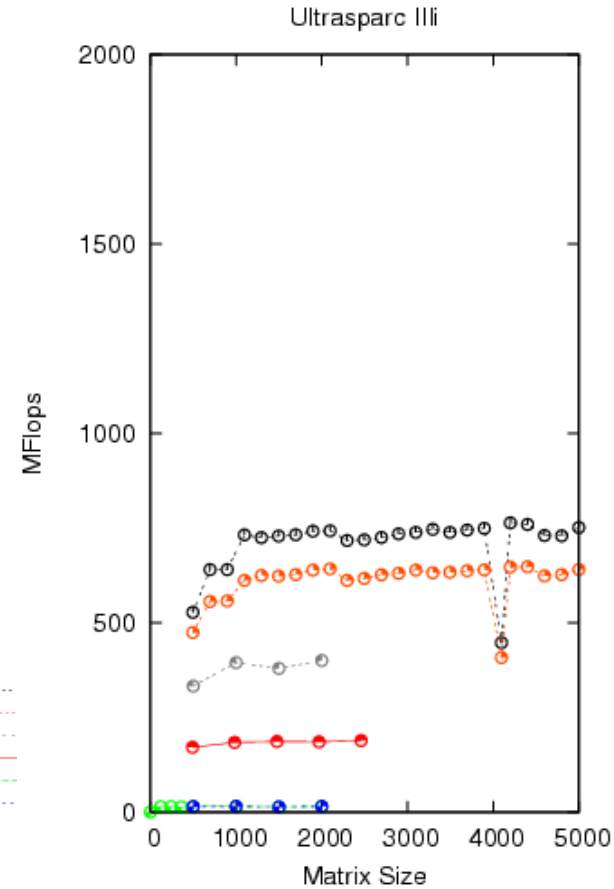
- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Perform Belady's register allocation on the basic block
  - Schedule using BRILA compiler



Control Structures

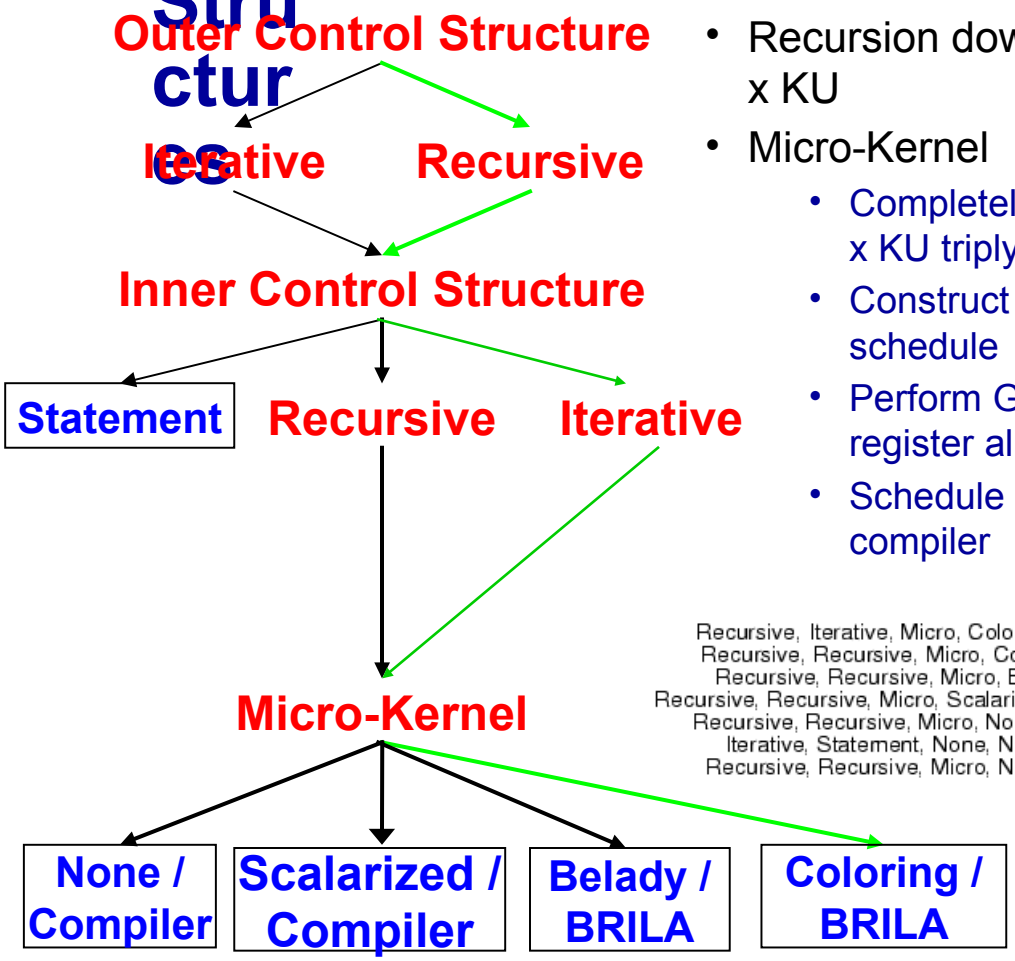


- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Construct a preliminary schedule
  - Perform Graph Coloring register allocation
  - Schedule using BRILA compiler

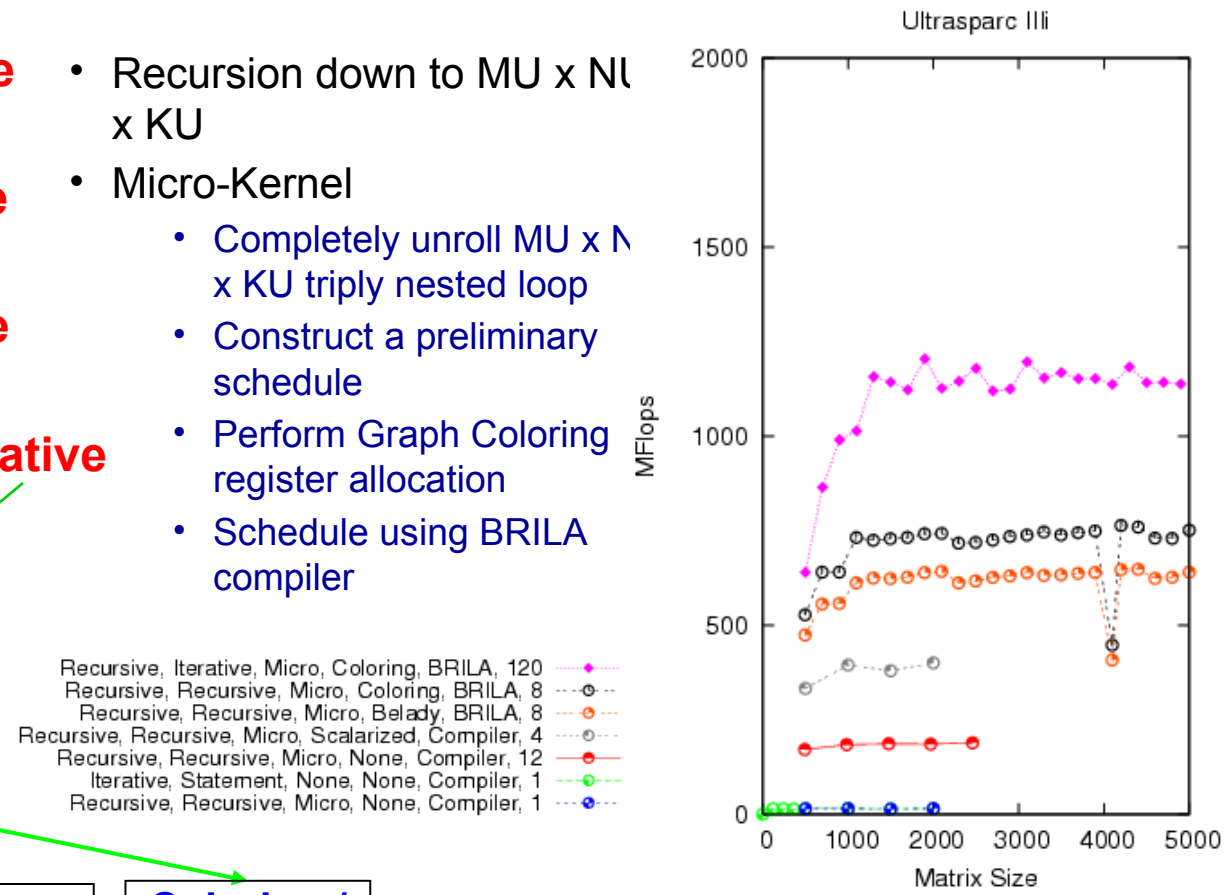


 Slide source: Roeder, Yotov, Pingali at Cornell/UTexas

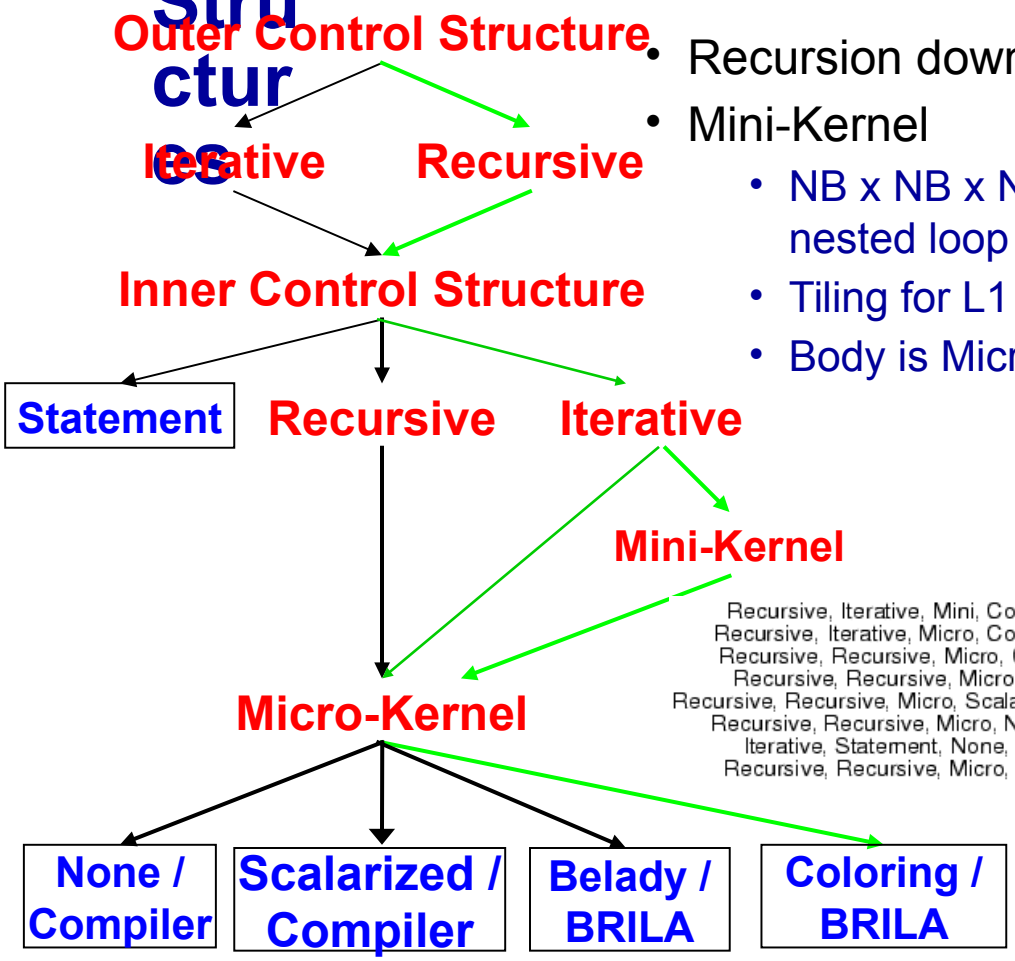
Control Structures



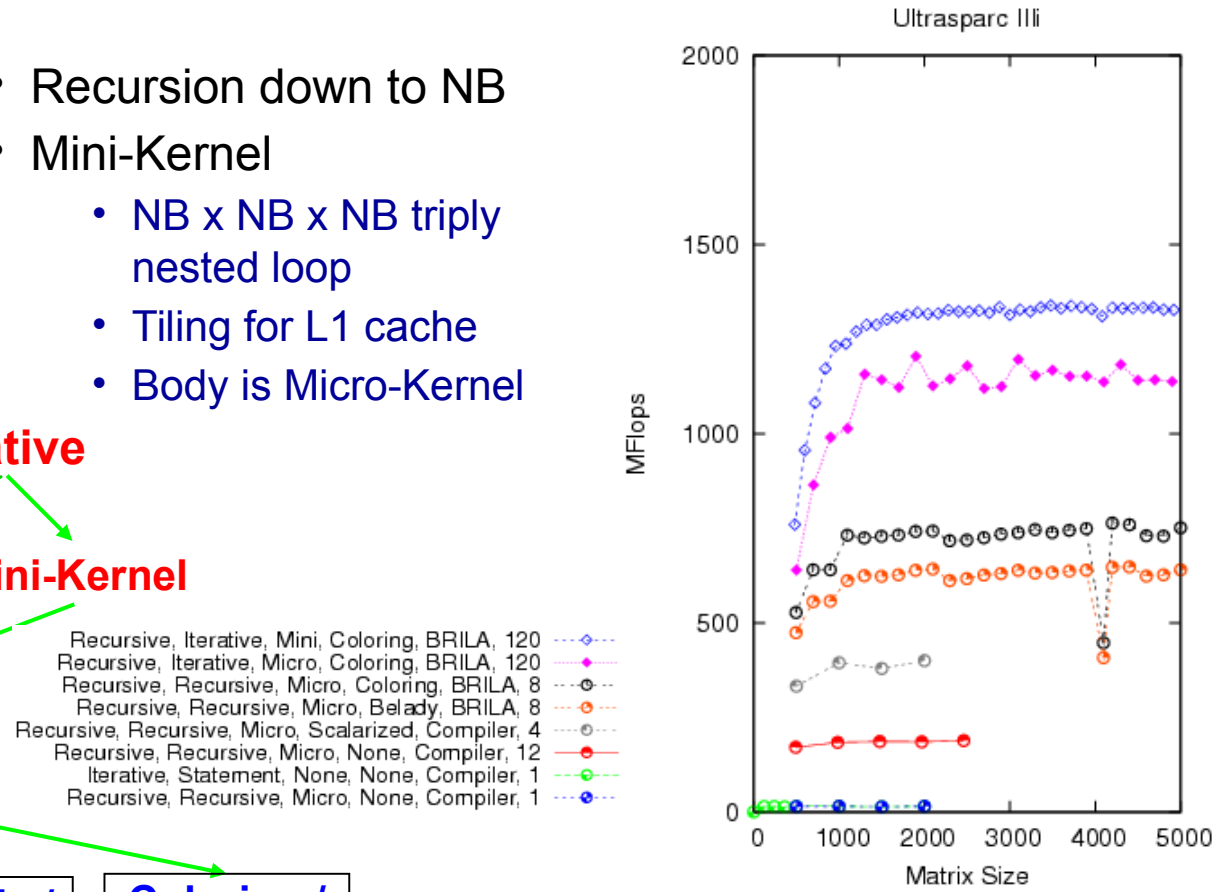
- Recursion down to MU x NU x KU
- Micro-Kernel
  - Completely unroll MU x NU x KU triply nested loop
  - Construct a preliminary schedule
  - Perform Graph Coloring register allocation
  - Schedule using BRILA compiler



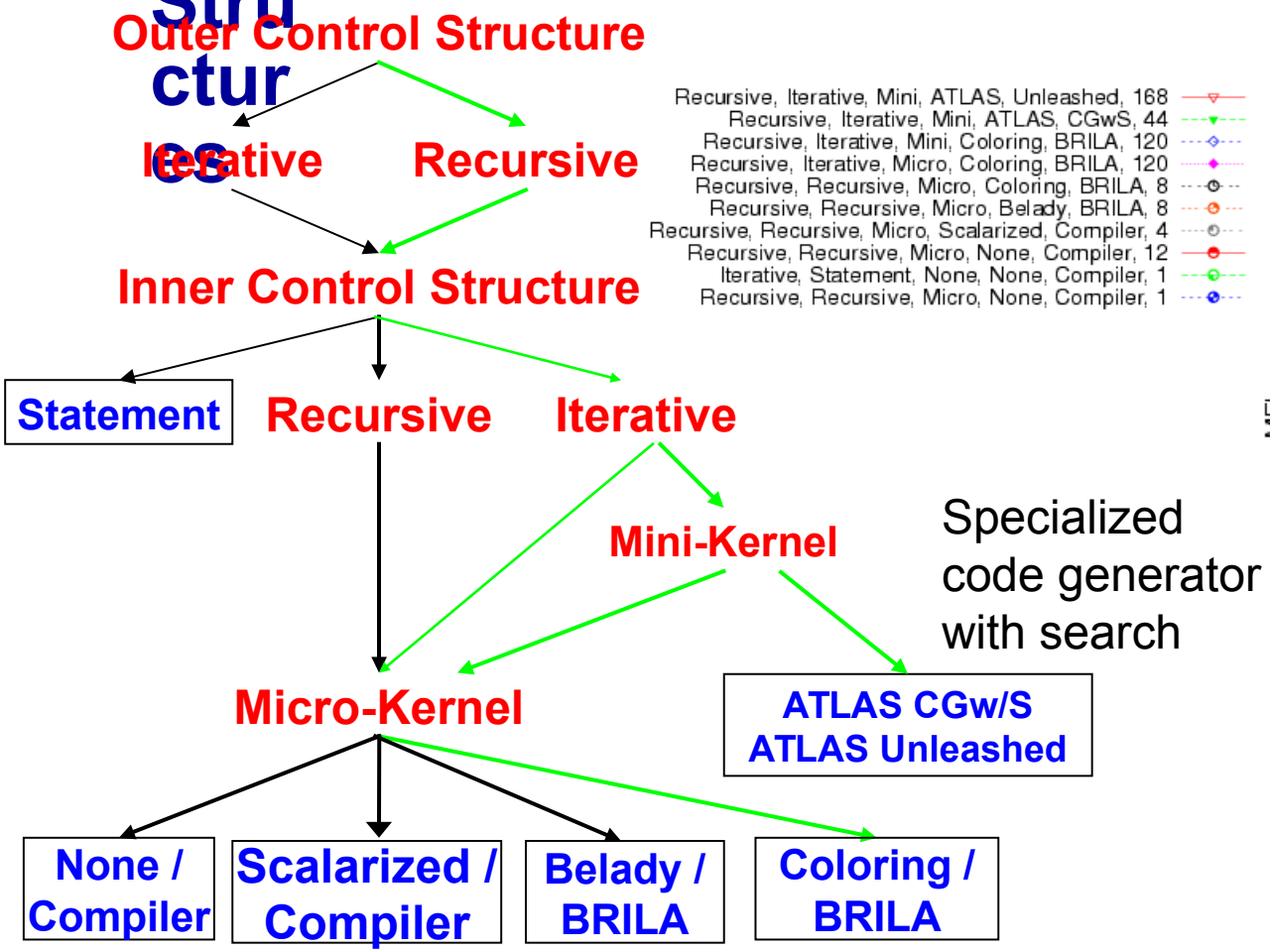
Control Structures



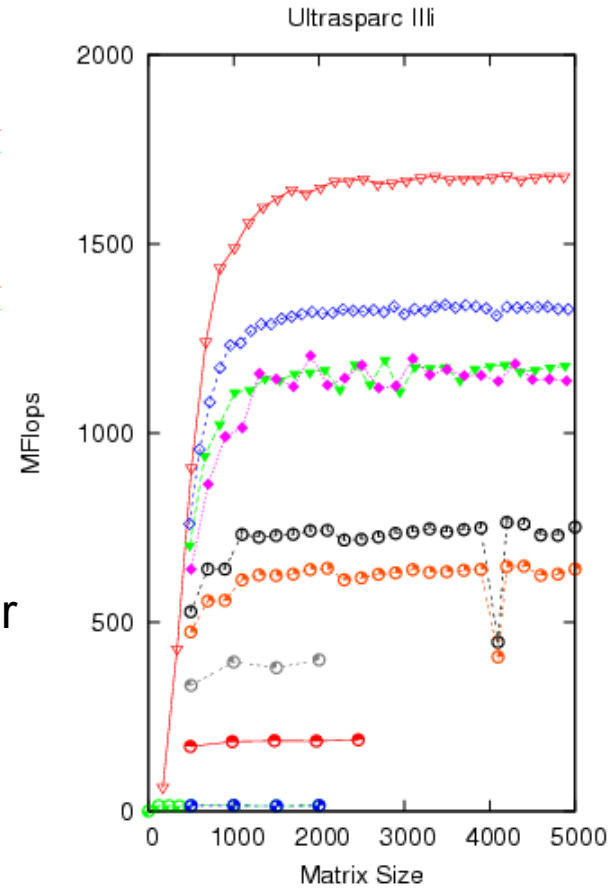
- Recursion down to NB
- Mini-Kernel
  - NB x NB x NB triply nested loop
  - Tiling for L1 cache
  - Body is Micro-Kernel



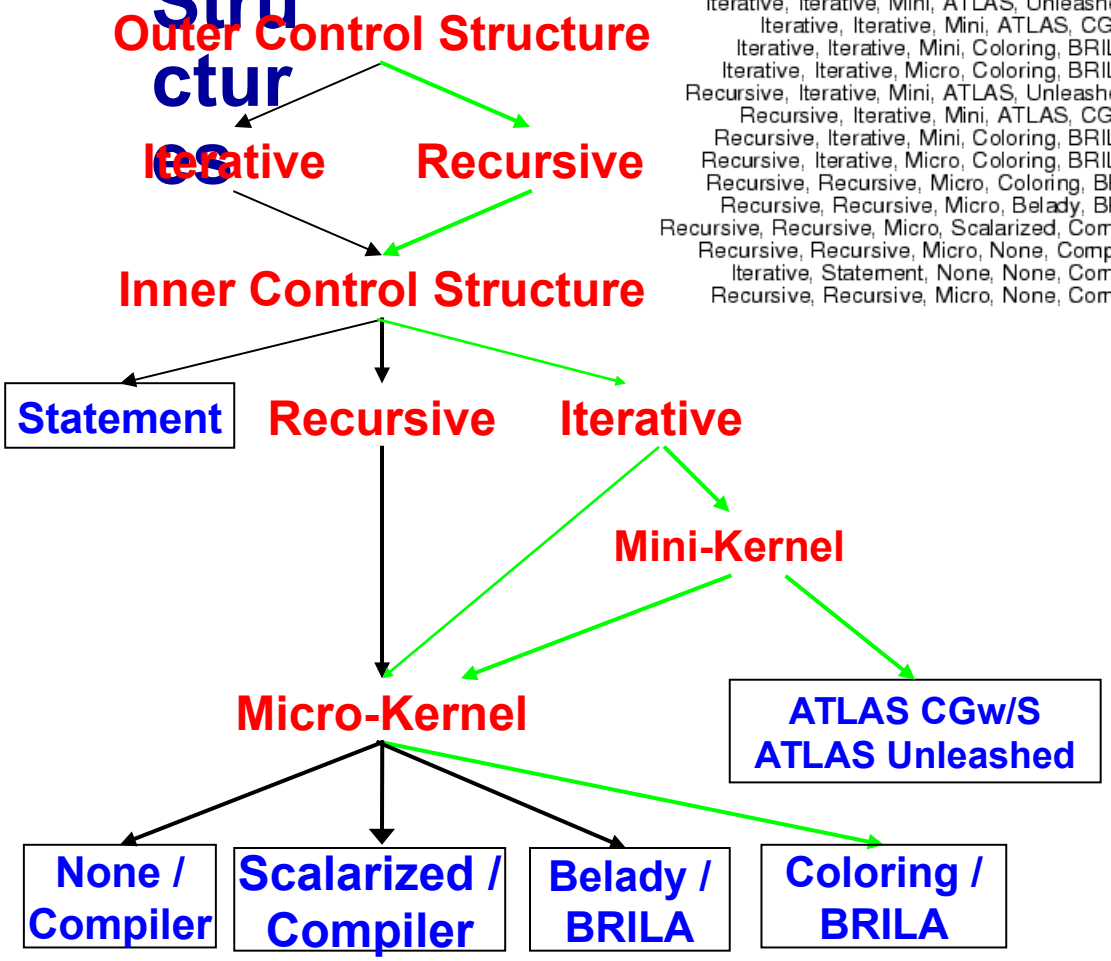
Control Structures

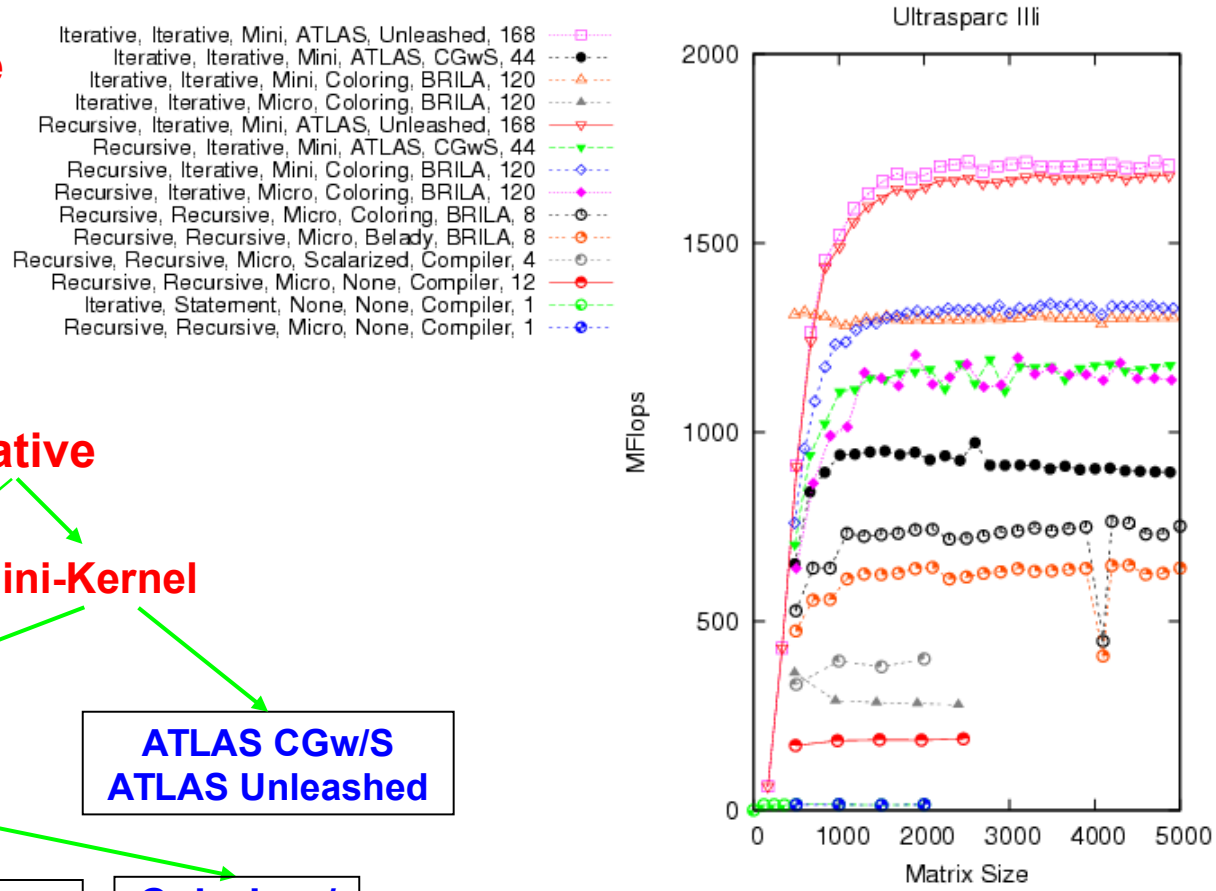


Specialized code generator with search



Control Structures





 Slide source: Roeder, Yotov, Pingali at Cornell/UTexas

Experience

- In practice, need to cut off recursion
- Implementing a high-performance Cache-Oblivious code is not easy
  - Careful attention to micro-kernel and mini-kernel is needed
- Using fully recursive approach with highly optimized recursive micro-kernel, Pingali et al report that they never got more than 2/3 of peak.
- Issues with Cache Oblivious (recursive) approach
  - Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Pre-fetching is needed to compete with best code: not well-understood in the context of CO codes

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 ($y=\alpha*x+y$), 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 <= 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}`
  - `More later in course`

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)

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 takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

Let $M = \begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix} = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$

Let p1 = (a12 - a22) * (b21 + b22)

p2 = (a11 + a22) * (b11 + b22)

p3 = (a11 - a21) * (b11 + b12)

p4 = (a11 + a12) * b22

p5 = a11 * (b12 - b22)

p6 = a22 * (b21 - b11)

p7 = (a21 + a22) * b11

Then m11 = p1 + p2 - p4 + p6

m12 = p4 + p5

m21 = p6 + p7

m22 = p2 - p3 + p5 - p7

Extends to nxn by divide&conquer

Strassen (continued)

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

- Asymptotically faster
  - Several times faster for large n in practice
  - Cross-over depends on machine
  - Available in several libraries
  - “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98
- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error

Other Fast Matrix Multiplication Algorithms

- Current world's record is $O(n^{2.376...})$ (Coppersmith & Winograd)
- Why does Hong/Kung theorem not apply?
- Possibility of $O(n^{2+\varepsilon})$ algorithm! (Cohn, Umans, Kleinberg, 2003)
- Fast methods (besides Strassen) may need unrealistically large n

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/`
  - PHiPAC: `www.icsi.berkeley.edu/~bilmes/phipac`
    in particular `tr-98-035.ps.gz`
  - ATLAS: `www.netlib.org/atlas`

Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies

```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d;
```

false read-after-write hazard 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

Original matrix
(numbers are addresses)

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

Reorganized into
2x2 blocks

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

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)
    - Weekly research meetings
    - Bebop.cs.berkeley.edu
    - About to release OSKI – tuning for sparse-matrix-vector multiply
  - 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 = # 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 possible on other data structures / algorithms