A Comparison of Cache-conscious and Cache-oblivious Codes

Tom Roeder, Cornell University
Kamen Yotov, IBM T. J. Watson (presenter)
Keshav Pingali, Cornell University

Joint work with Fred Gustavson and John Gunnels (IBM T. J. Watson)

Motivation

- Current compilers do not generate good code even for BLAS
  - Conjecture: code generation requires parameters like tile sizes and loop unroll factors whose optimal values are difficult to determine using analytical models
- Previous work on ATLAS
  - Actually, compiler models are quite adequate to produce optimized iterative cache-conscious MMM (Yotov et al. 2005)
  - So what is going on inside compilers?
    - Hard to know because compilers like XLF are not in public domain
- Goal
  - Build a domain-specific compiler (BRILA) for dense linear algebra programs
  - Input
    - High-level block-recursive algorithms
    - Key data structure is matrix, not array
  - Output
    - Code optimized for memory hierarchy
- Question:
  - What should output look like?

Two Possible Answers

- Cache-oblivious (CO) approach
  - Execute recursive algorithm directly
    - Not aware of memory hierarchy: approximate blocking
    - I/O optimal
  - Used in FFT implementations, e.g. FFTW
    - Little data reuse
- Cache-conscious (CC) approach:
  - Execute carefully blocked iterative algorithm
    - Code (and data structures) have parameters that depend on memory hierarchy
  - Used in dense linear algebra domain, e.g. BLAS, LAPACK
    - Lots of data reuse
- Questions
  - How does performance of CO approach compare with that of CC approach for algorithms with a lot of reuse such as dense LA?
  - More generally, under what assumptions about hardware and algorithms does CO approach perform well?

Organization of Talk

- Non-standard view of blocking
  - Reduce bandwidth required from memory
- CO and CC approaches to blocking
  - Control structures
  - Data structures
- Experimental results
  - UltraSPARC IIIi
  - Itanium
  - Xeon
  - Power 5
- Lessons and ongoing work

Blocking

- Microscopic view
  - Blocking reduces latency of a memory access
- Macroscopic view
  - Memory hierarchy can be ignored if
    - memory has enough bandwidth to feed processor
    - data can be pre-fetched to hide memory latency
  - Blocking reduces bandwidth needed from memory
- Useful to consider macroscopic view in more detail

Example: MMM on Itanium 2

- Processor features
  - 2 FMAs per cycle
  - 126 effective FP registers
- Basic MMM

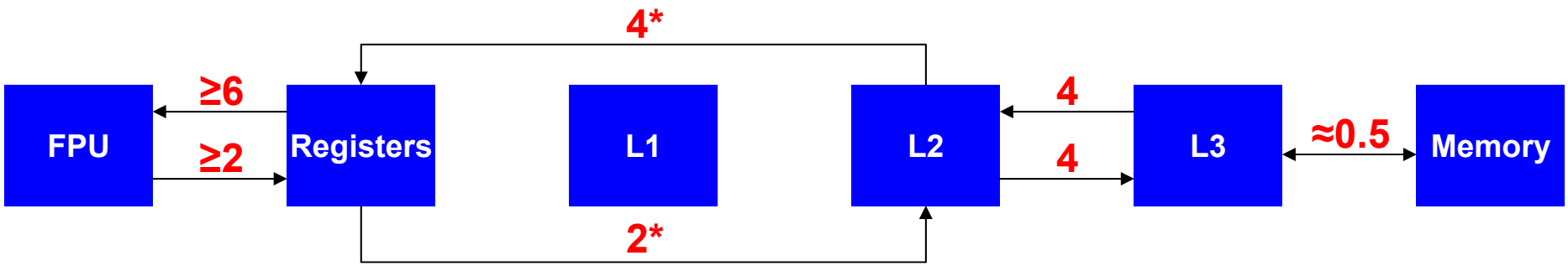
```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      C[i, j] += A[i, k] * B[k, j];
```

- Execution requirements
  - N^3 multiply-adds
    - Ideal execution time = N^3 / 2 cycles
  - 3 N^3 loads + N^3 stores = 4 N^3 memory operations
- Bandwidth requirements
  - 4 N^3 / (N^3 / 2) = 8 doubles / cycle
- Memory cannot sustain this bandwidth but register file can

Reduce Bandwidth by Blocking

- Square blocks: NB x NB x NB
  - working set must fit in cache
  - size depends on schedule, maximum is 3 NB^2
- Data touched (doubles)
  - Block: 4 NB^2
  - Total: $(N / NB)^3 * 4\ NB^2 = 4\ N^3 / NB$
- Ideal execution time (cycles)
  - $N^3 / 2$
- Required bandwidth from memory (doubles per cycle)
  - $(4\ N^3 / NB) / (N^3 / 2) = 8 / NB$
- General picture for multi-level memory hierarchy
  - Bandwidth required from level L+1 = $8 / NB_L$
- Constraints
  - Lower bound: $8 / NB_L \leq$ Bandwidth between L and L+1
  - Upper bound: Working set of block computation ≤ Capacity(L)

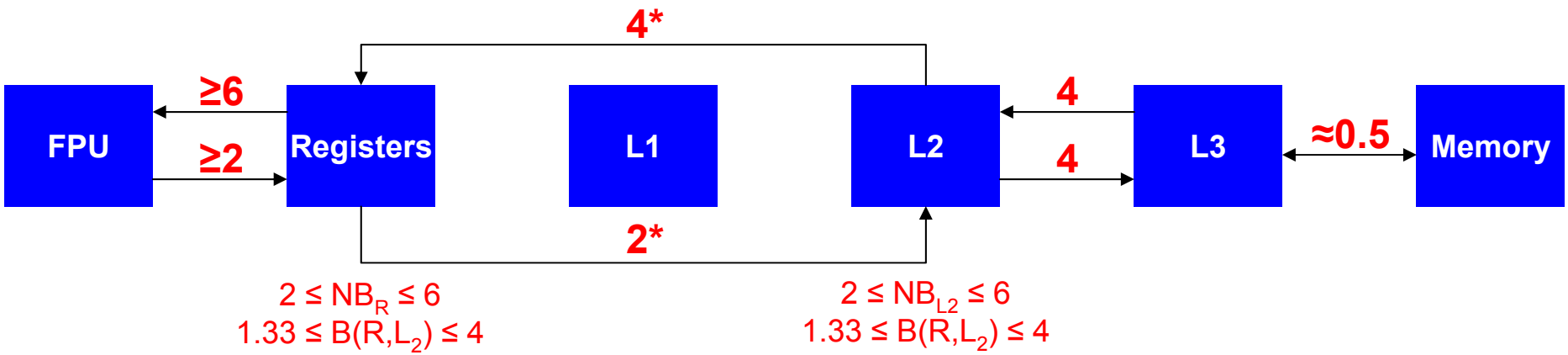
Example: MMM on Itanium 2



* Bandwidth in doubles per cycle; Limit 4 accesses per cycle between registers and L2

- Between Register File and L2
  - Constraints
    - $8 / NB_R \leq 4$
    - $3 * NB_R^2 \leq 126$
  - Therefore Bandwidth(R,L2) is enough for $2 \leq NB_R \leq 6$
    - $NB_R = 2$ required $8 / NB_R = 4$ doubles per cycle from L2
    - $NB_R = 6$ required $8 / NB_R = 1.33$ doubles per cycle from L2
    - $NB_R > 6$ possible with better scheduling

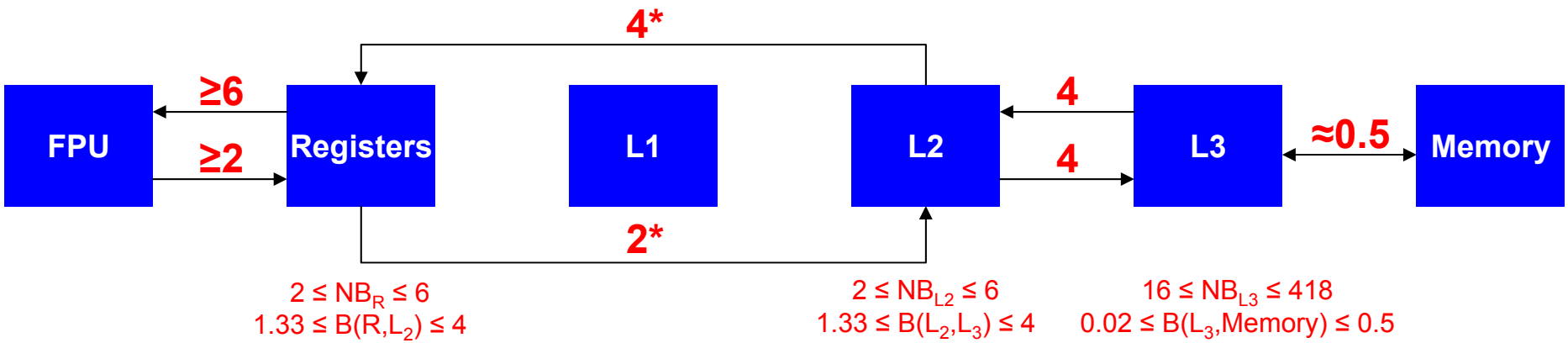
Example: MMM on Itanium 2



* Bandwidth in doubles per cycle; Limit 4 accesses per cycle between registers and L2

- Between L2 and L3
  - Sufficient bandwidth without blocking at L2

Example: MMM on Itanium 2



* Bandwidth in doubles per cycle; Limit 4 accesses per cycle between registers and L2

- Between L3 and Memory
  - Constraints
    - $8 / NB_{L3} \leq 0.5$
    - $3 * NB_{L3}^2 \leq 524288$ (4MB)
  - Therefore Bandwidth(L3,Memory) is enough for $16 \leq NB_{L3} \leq 418$
    - $NB_{L3} = 16$ required $8 / NB_{L3} = 0.5$ doubles per cycle from Memory
    - $NB_{L3} = 418$ required $8 / NB_R \approx 0.02$ doubles per cycle from Memory
    - $NB_{L3} > 418$ possible with better scheduling

Lessons

- Blocking
  - Useful to reduce bandwidth requirements
- Block size
  - Does not have to be exact
  - Enough to lie within an interval
  - Interval depends on hardware parameters
  - Approximate blocking may be OK
- Latency
  - Use pre-fetching to reduce expected latency

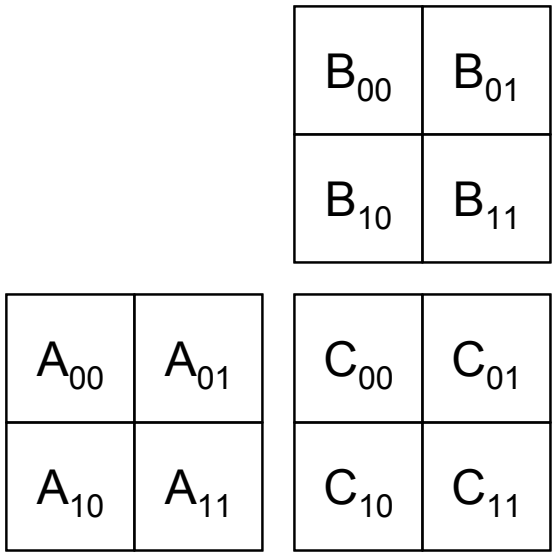
Organization of Talk

- Non-standard view of blocking
  - Reduce bandwidth required from memory
- CO and CC approaches to blocking
  - Control structures
  - Data structures
- Experimental results
  - UltraSPARC IIIi
  - Itanium
  - Xeon
  - Power 5
- Lessons and ongoing work

Implementation of Blocking

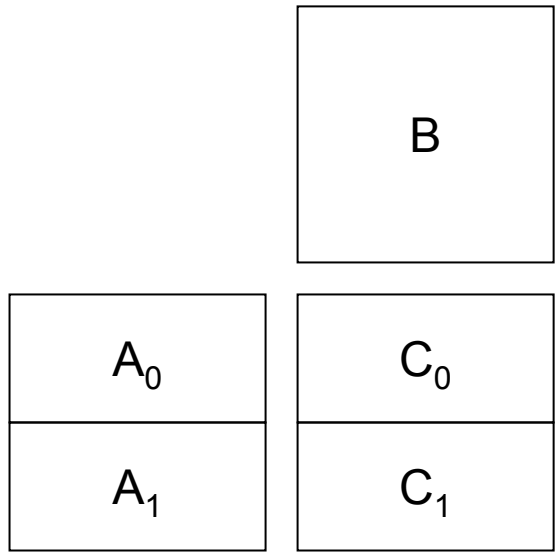
- Control structure
  - What are the block computations?
  - In what order are they performed?
  - How is this order generated?
- Data structure
  - Non-standard storage orders to match control structure

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.

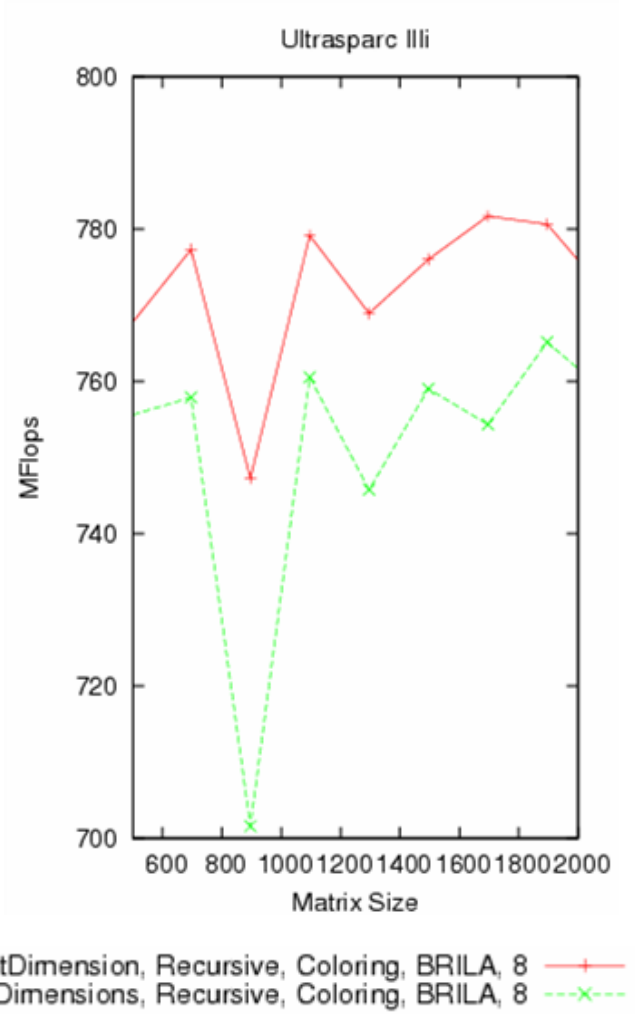


$$C_0 = A_0*B$$
$$C_1 = A_1*B$$

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

Cache-Oblivious: Discussion

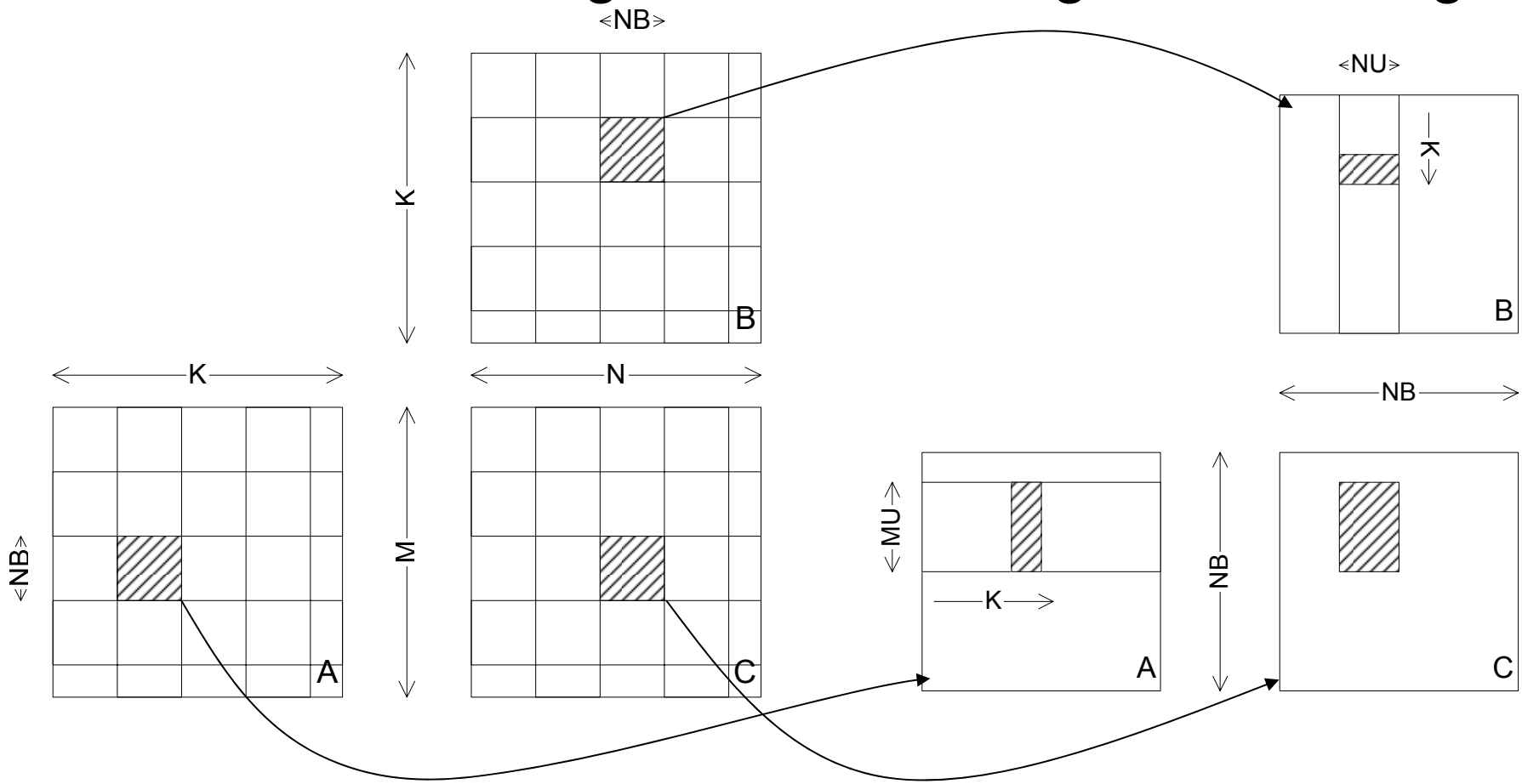
- Block sizes
  - Generated dynamically at each level in the recursive call tree
- Our experience
  - Performance is similar
  - Use AD for the rest of the talk



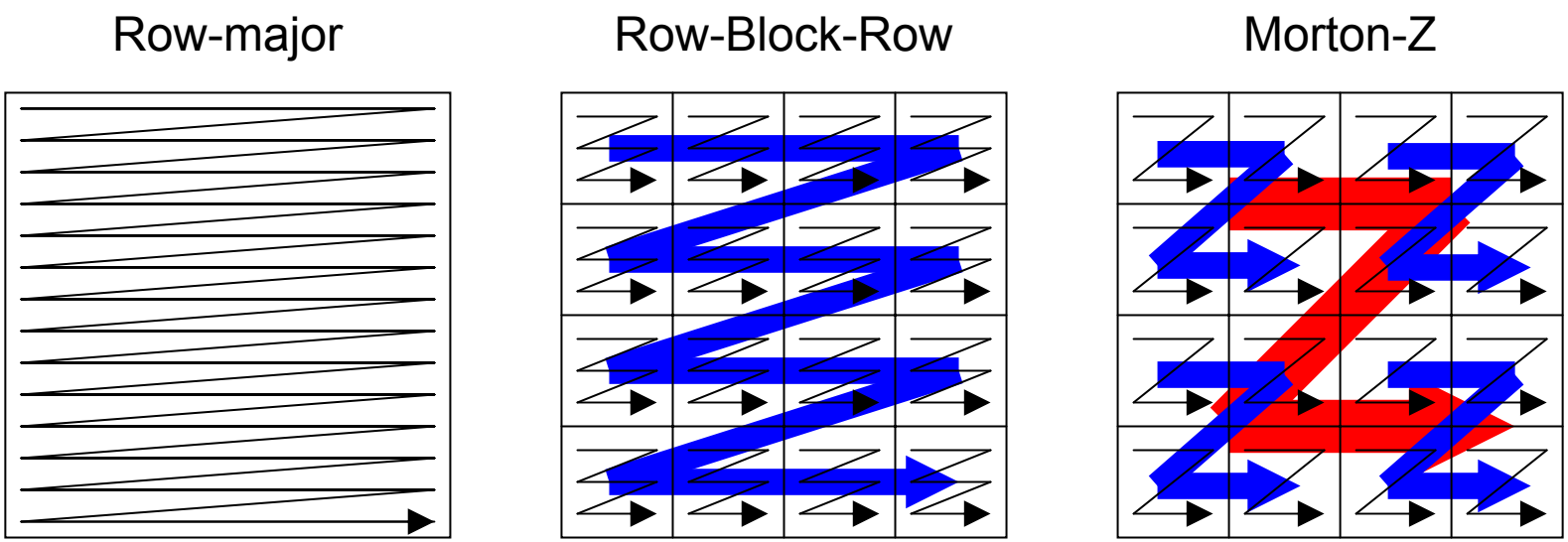
Cache-Conscious Algorithms

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

Structure of CC Code



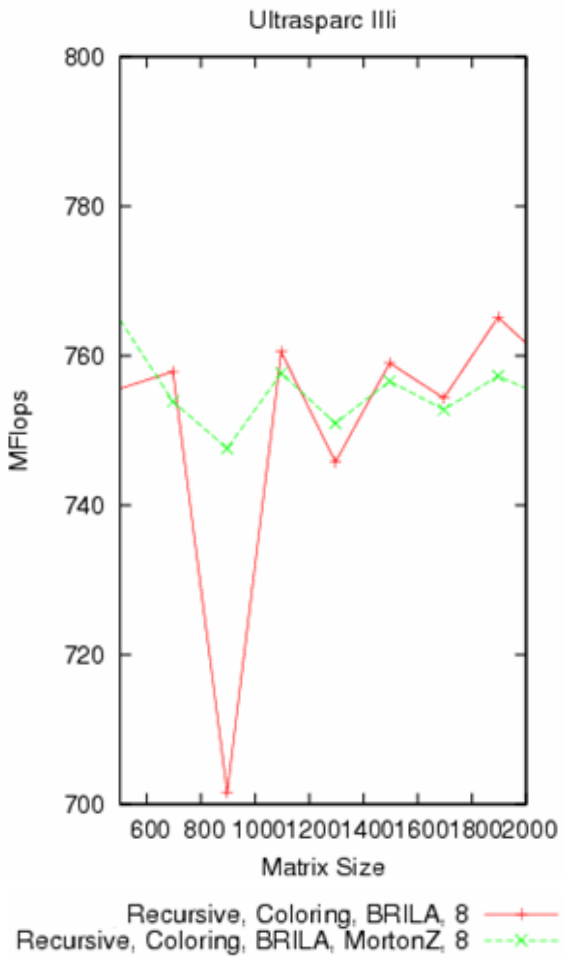
Data Structures



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

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



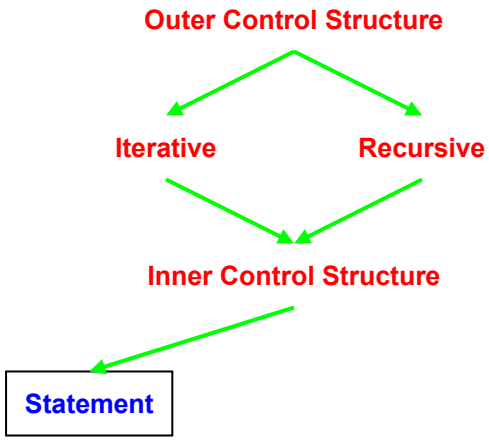
Organization of Talk

- Non-standard view of blocking
  - Reduce bandwidth required from memory
- CO and CC approaches to blocking
  - Control structures
  - Data structures
- Experimental results
  - UltraSPARC IIIi
  - Itanium
  - Xeon
  - Power 5
- Lessons and ongoing work

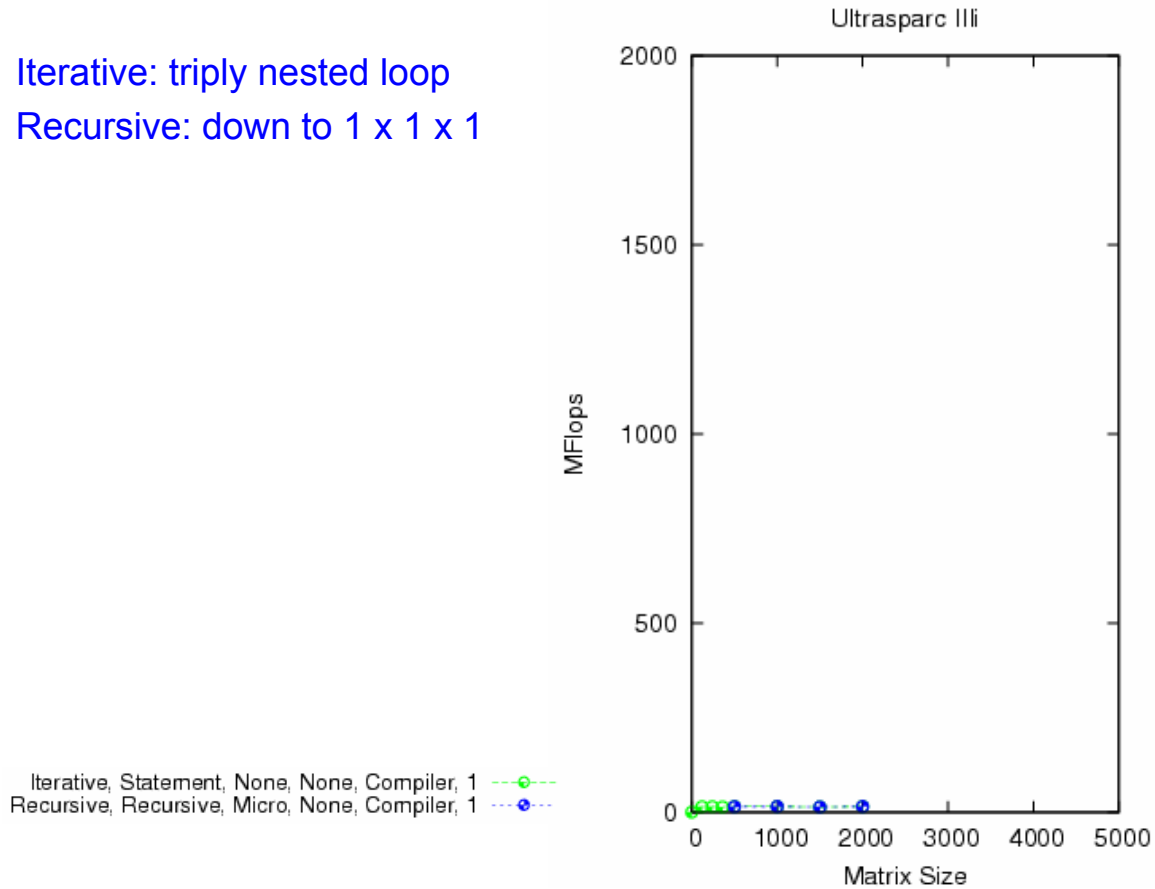
UltraSPARC IIIi

- 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

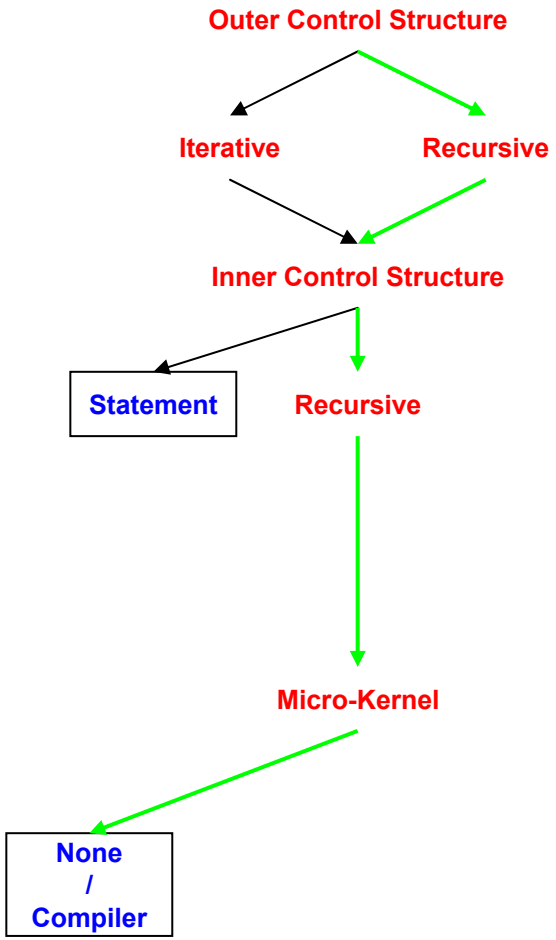
Control Structures



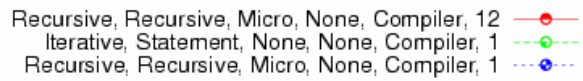
- Iterative: triply nested loop
- Recursive: down to 1 x 1 x 1

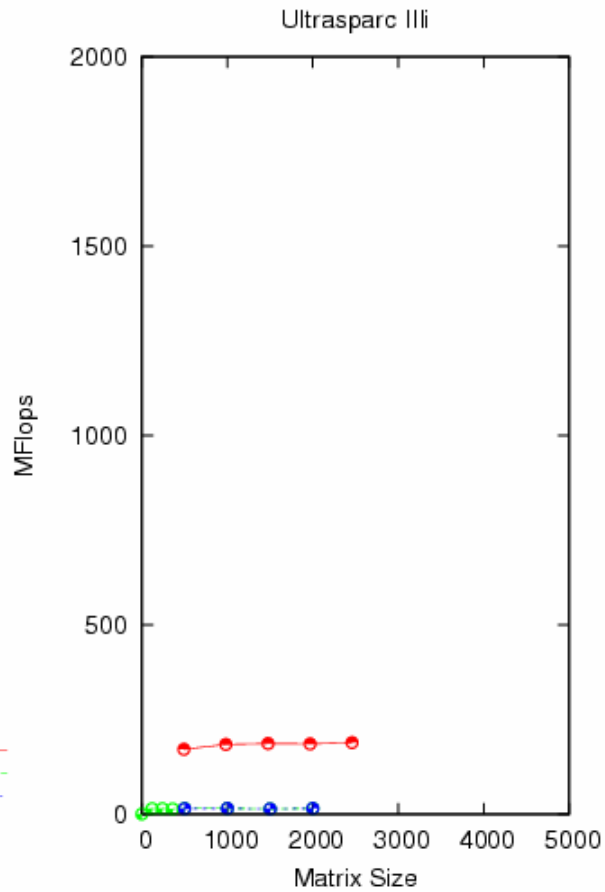


Control Structures

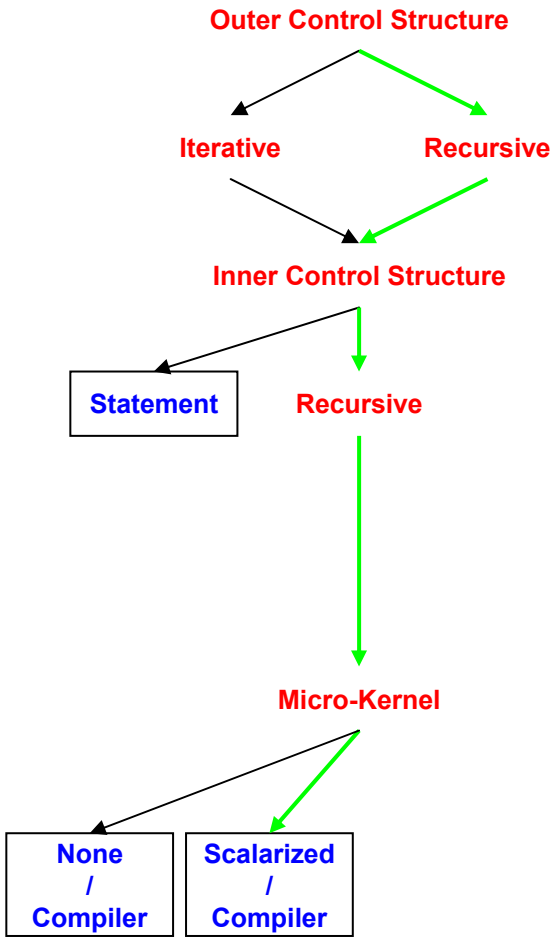


- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel:
  - The basic block compiled with native compiler
- Best performance for NB =12
- Compiler unable to use registers
- Unfolding reduces control overhead
  - limited by I-cache

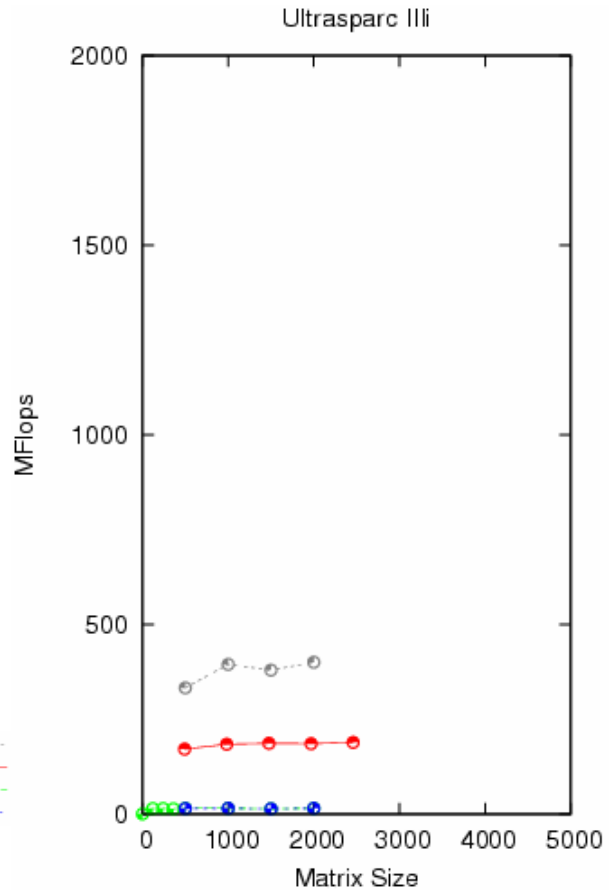




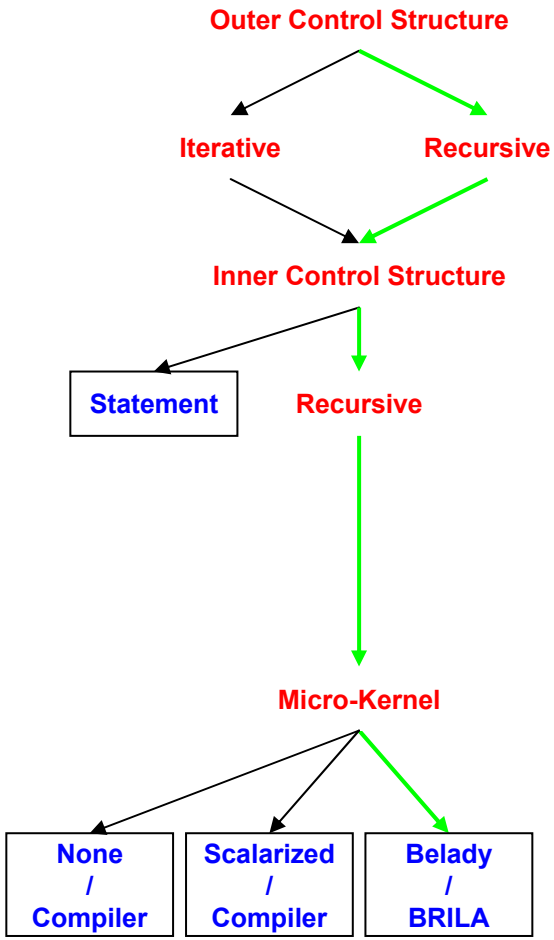
Control Structures



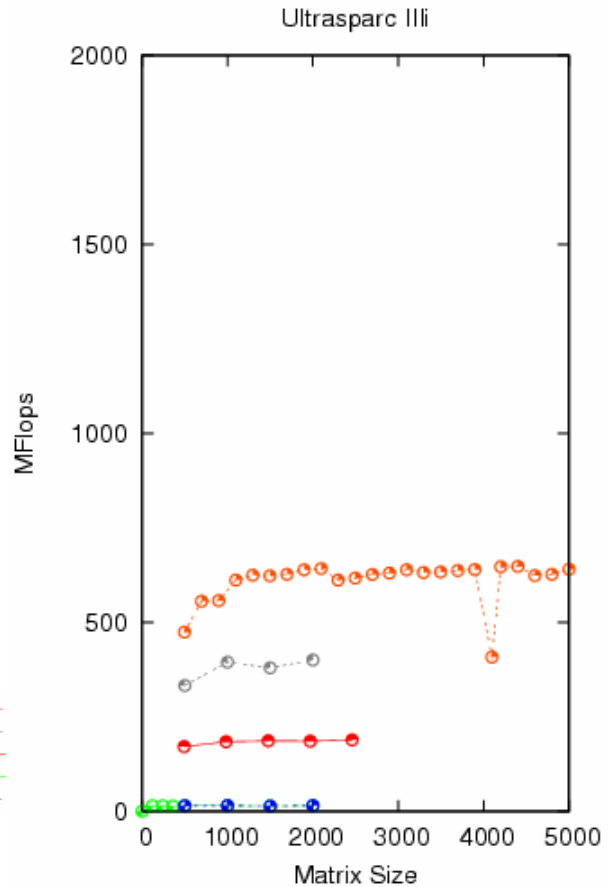
- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Scalarize all array references in the basic block
  - Compile with native compiler



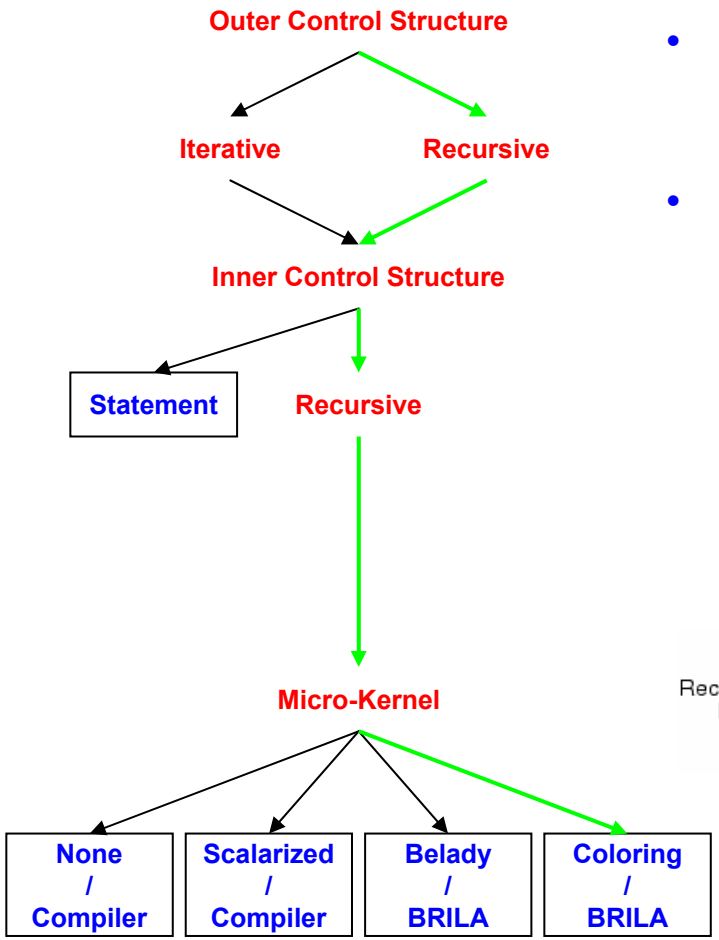
Control Structures



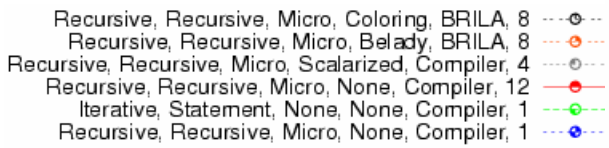
- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Perform Belady's register allocation on the basic block
  - Schedule using BRILA compiler

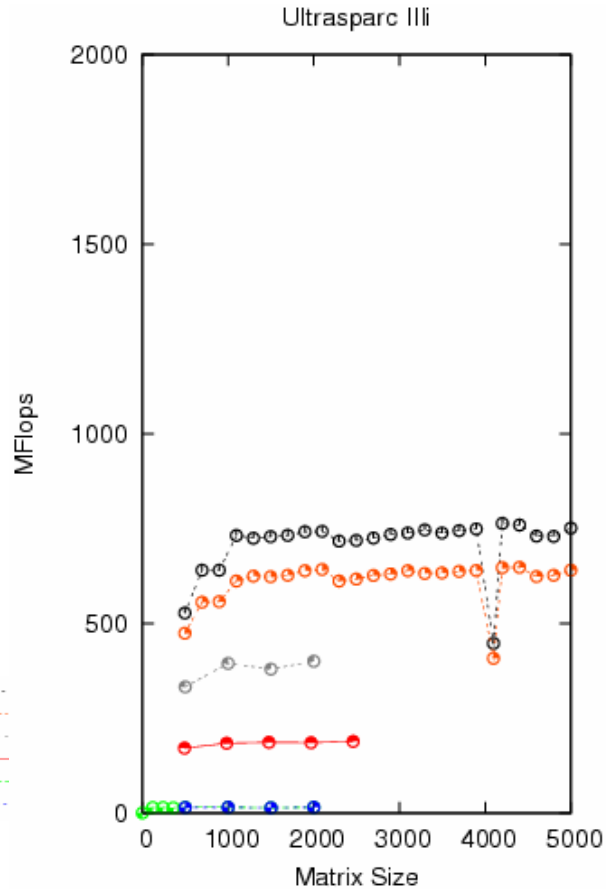


Control Structures

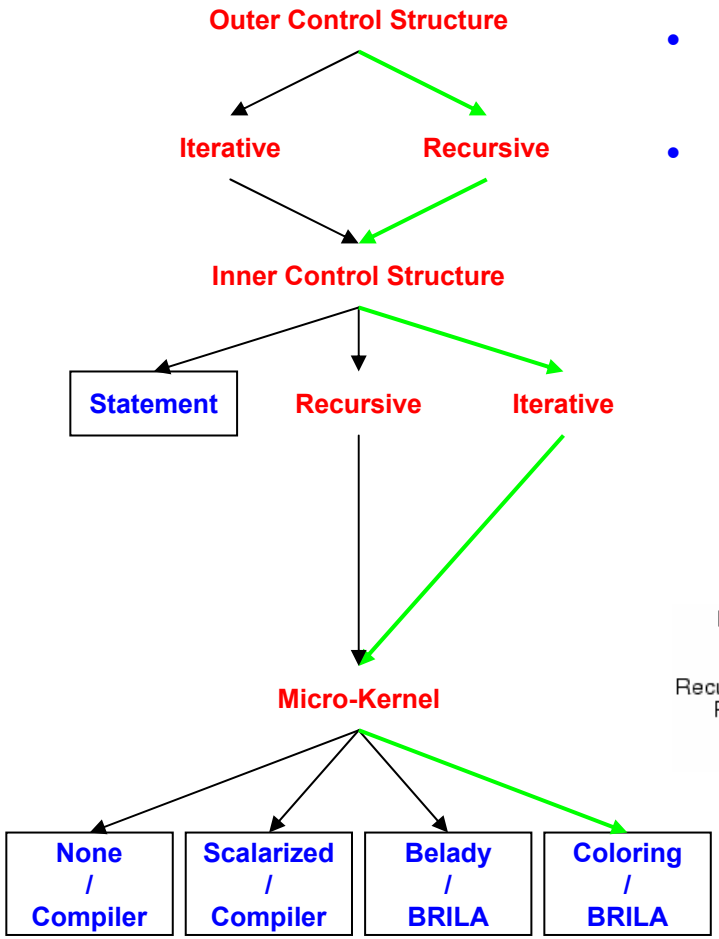


- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
  - Construct a preliminary schedule
  - Perform Graph Coloring register allocation
  - Schedule using BRILA compiler

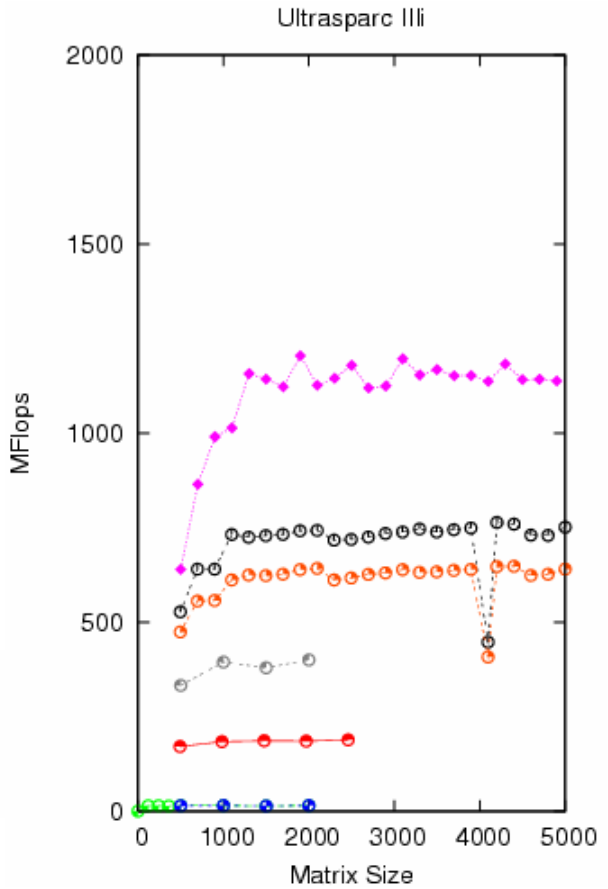




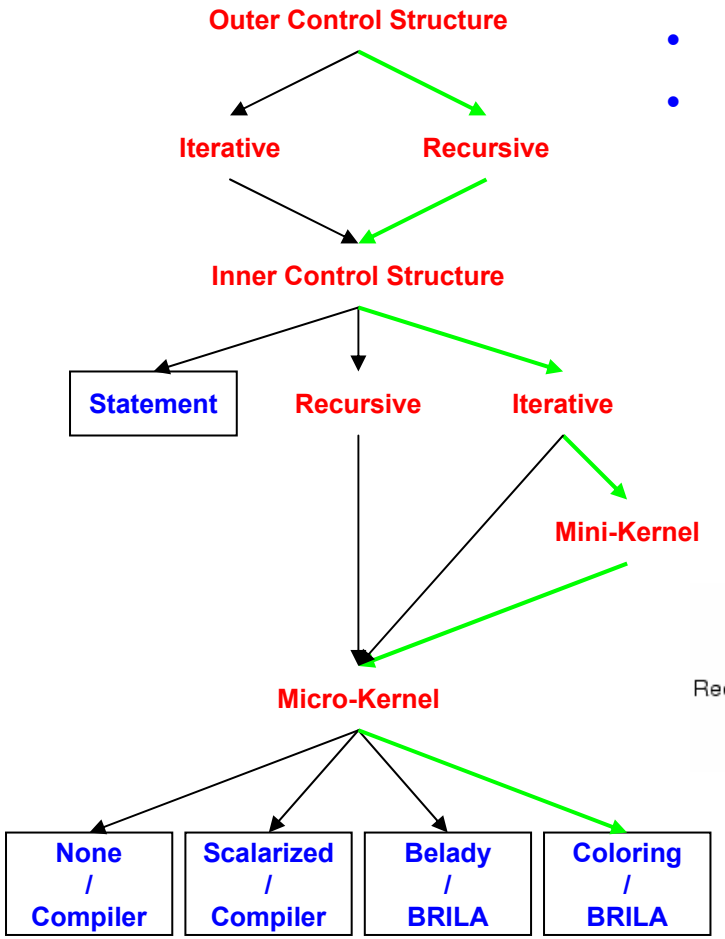
Control Structures



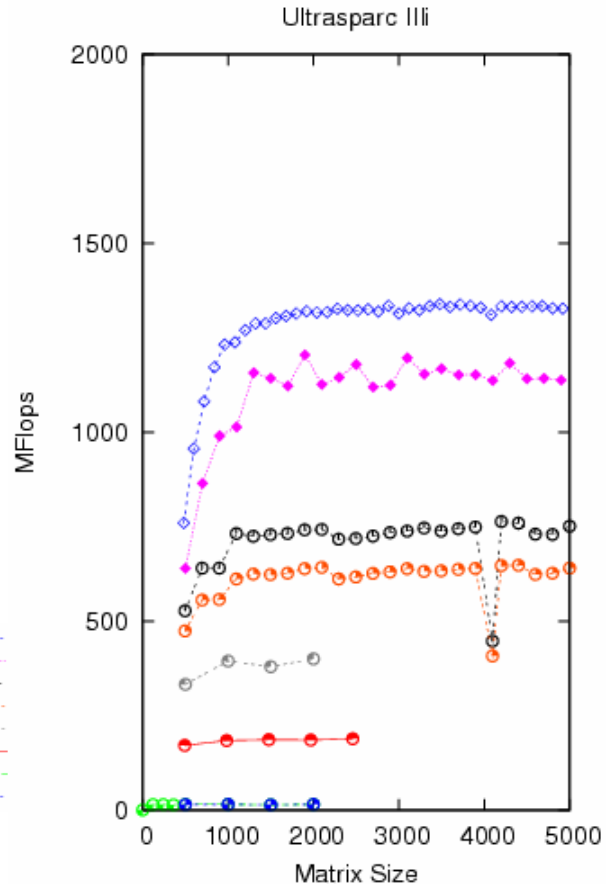
- Recursion down to MU x NU x KU
- Micro-Kernel
  - Completely unroll MU x NU x KU triply nested loop
  - Construct a preliminary schedule
  - Perform Graph Coloring register allocation
  - Schedule using BRILA compiler



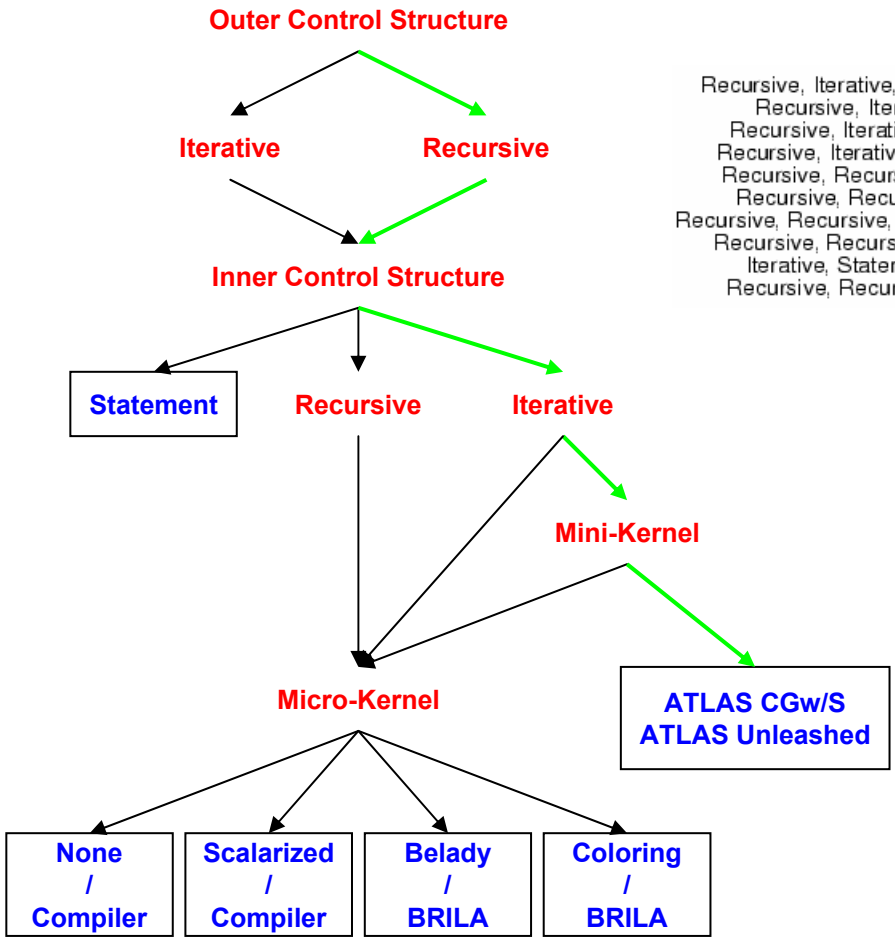
Control Structures

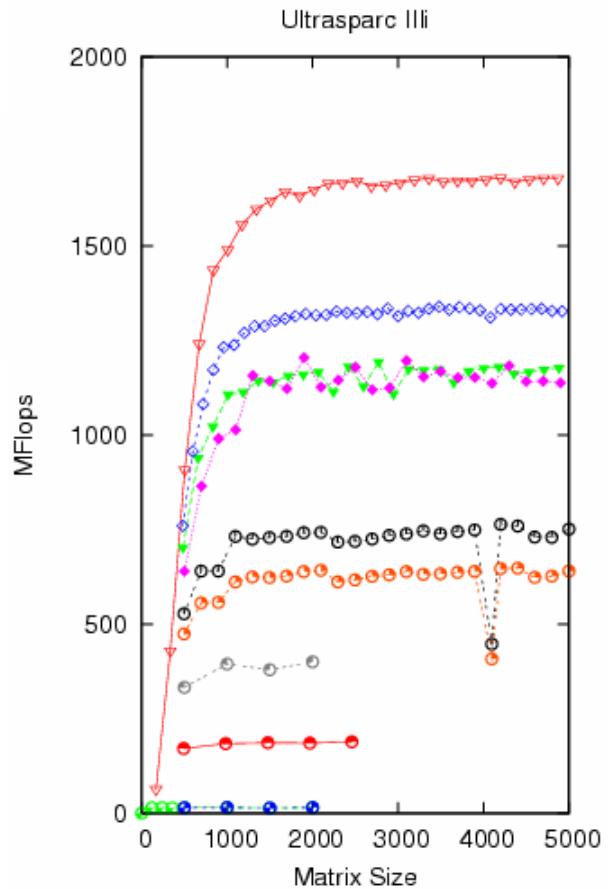


- Recursion down to NB
- Mini-Kernel
  - NB x NB x NB triply nested loop
  - Tiling for L1 cache
  - Body is Micro-Kernel

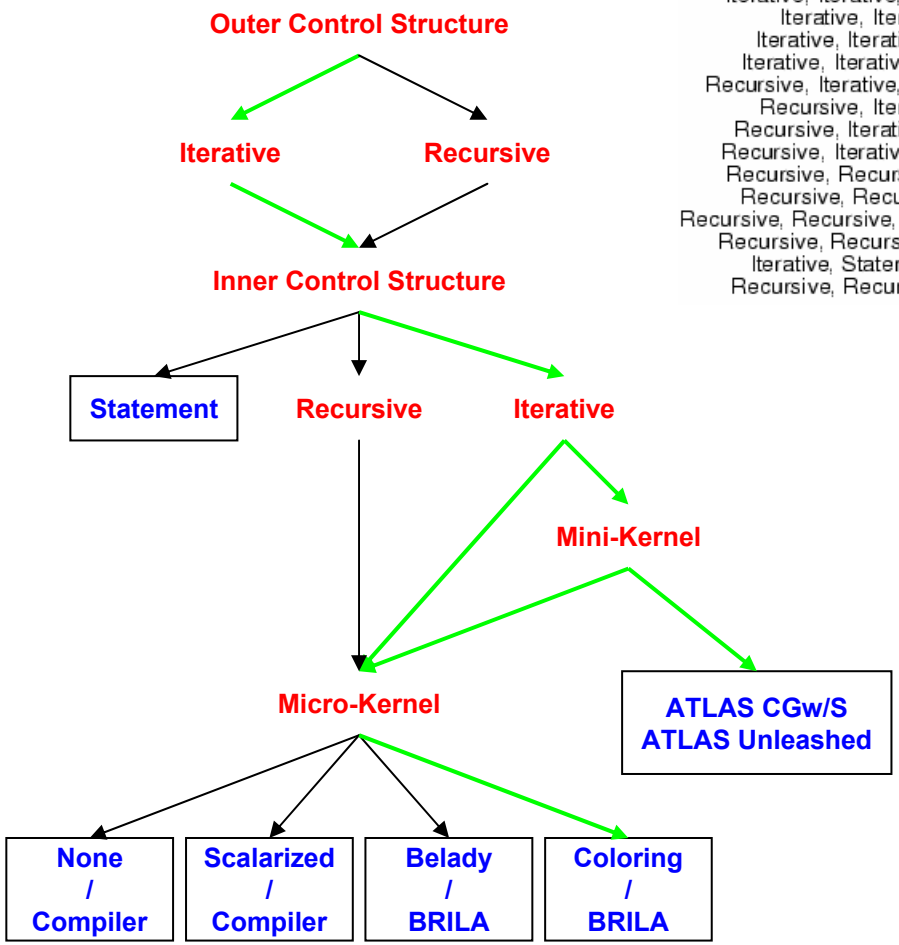


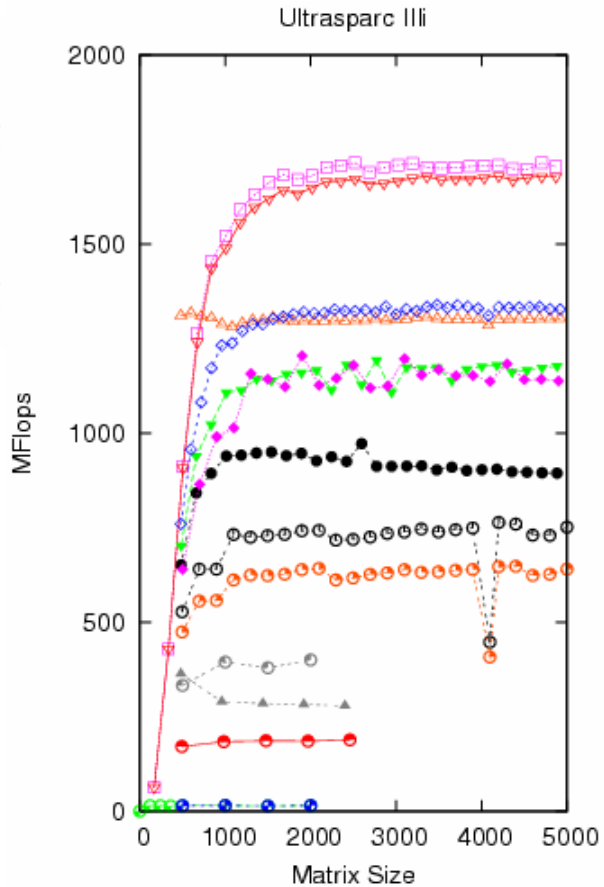
Control Structures



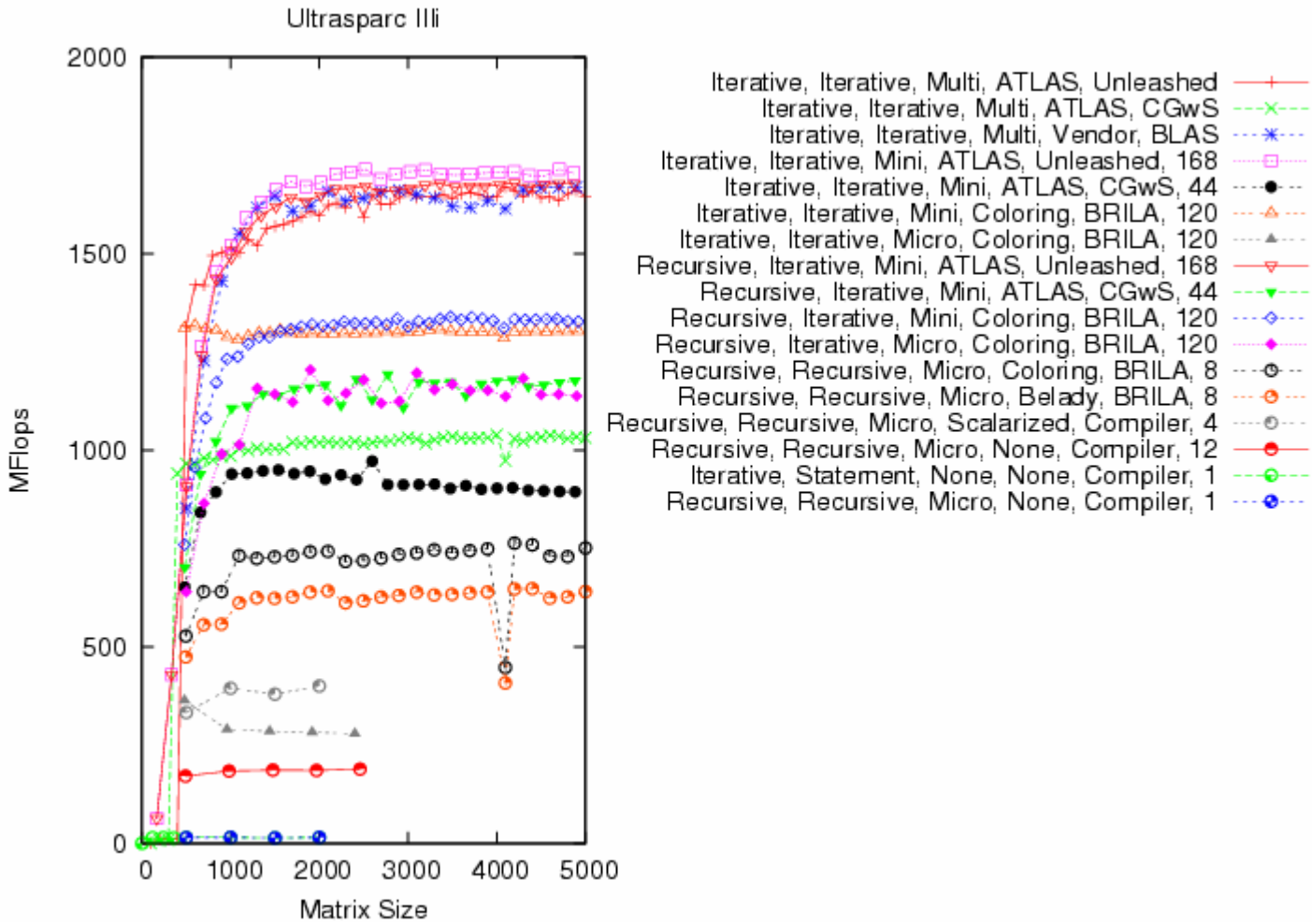


Control Structures





UltraSPARC IIIi Complete



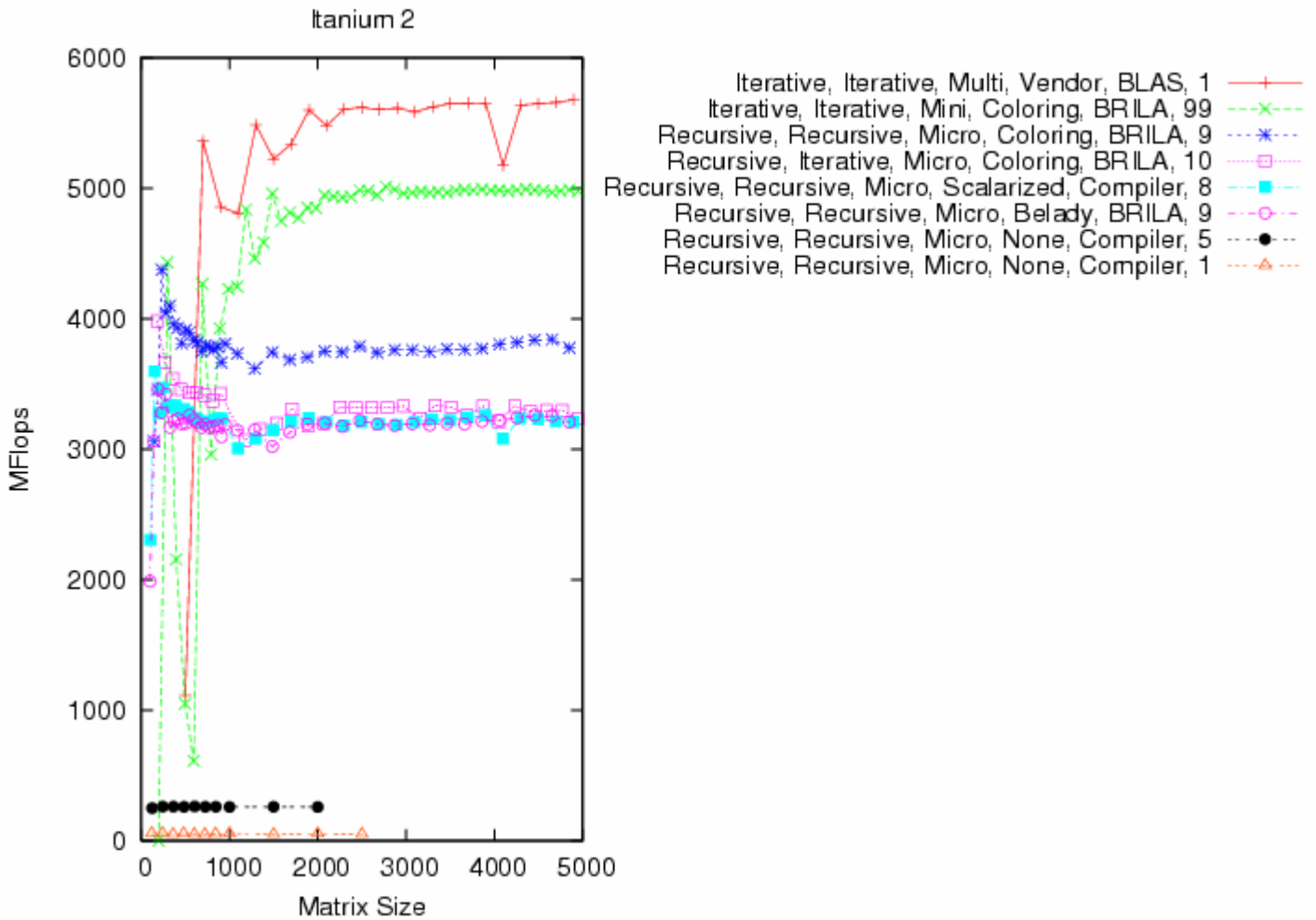
Some Observations

- Iterative has been proven to work well in practice
  - Vendor BLAS, ATLAS, etc.
  - But requires a lot of work to produce code and tune parameters
- Implementing a high-performance CO code is not easy
  - Careful attention to micro-kernel and mini-kernel is needed
- Recursive suffers overhead on several fronts
  - Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Recursive Micro-Kernels have large code size, which sometimes impacts instruction cache performance
  - Obtaining high-performance from recursive outer structure requires large kernels at the leaves to reduce recursive overhead
  - Using fully recursive approach with highly optimized micro-kernel, we never got more than 2/3 of peak.
  - We are just starting analyze the numbers
- Automating code generation:
  - Pre-fetching in iterative codes can be automated
  - Not obvious how to do it for CO codes

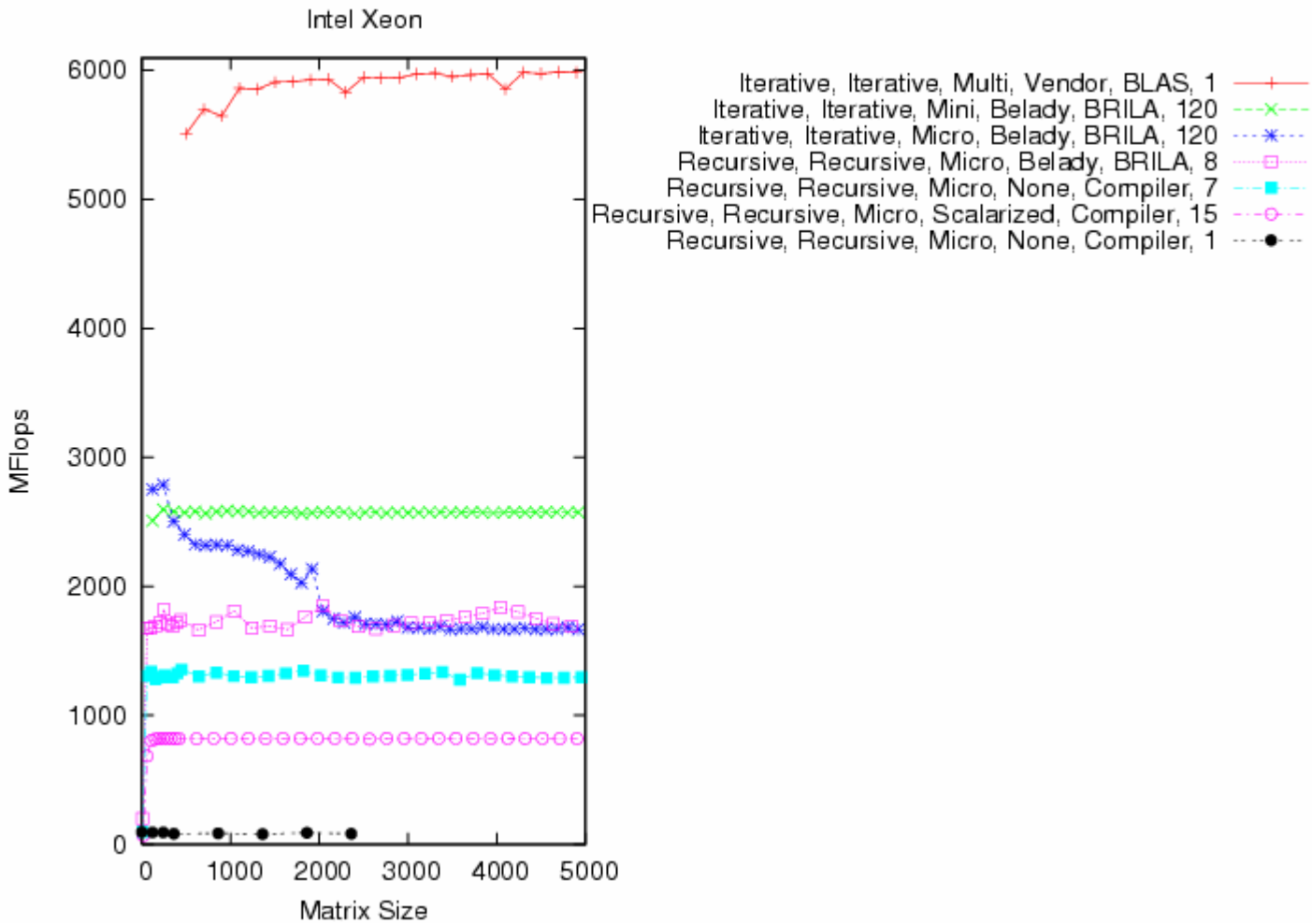
Ongoing Work

- Explain performance of all results shown
- Complete ongoing Matrix Transpose study
- I/O optimality
  - Interesting theoretical results for simple model of computation
  - What additional aspects of hardware/program need to be modeled for it to be useful in practice?
- Compiler-generated iterative multi-level blocking for dense linear algebra programs
  - BRILA Compiler

Itanium 2 (In)Complete



Xeon (In)Complete



Power 5 (In)Complete

- In the works…