A Crash Course in Compilers for Parallel Computing

Mary Hall

Fall, 2008

Overview of "Crash Course"

- L1: Data Dependence Analysis and Parallelization (Oct. 30)
- L2 & L3: Loop Reordering Transformations, Reuse Analysis and Locality Optimization (Nov. 6)
- L4: Autotuning Compiler Technology (Nov. 13)

Outline of Lecture

I. Summary of Previous Weeks
   - What will we use today?
II. Motivation
III. Autotuning for Locality in ATLAS
IV. Generalized Autotuning Compiler
   - On application code
   - Discussion of SSE and Multi-core
   - Empirical search
V. Potential Future Research Directions

Equivalence to Integer Programming

- Need to determine if F(i) = G(i'), where i and i' are iteration vectors, with constraints i,i' >= L, U>= i, i'
- **Example:**

```
for (i=2; i<=100; i++)
    A[i] = A[i-1];
```

- **Inequalities:**

0 <= i1 <= 100, i2 = i1 - 1, i2 <= 100

integer vector I, *AI <= b*

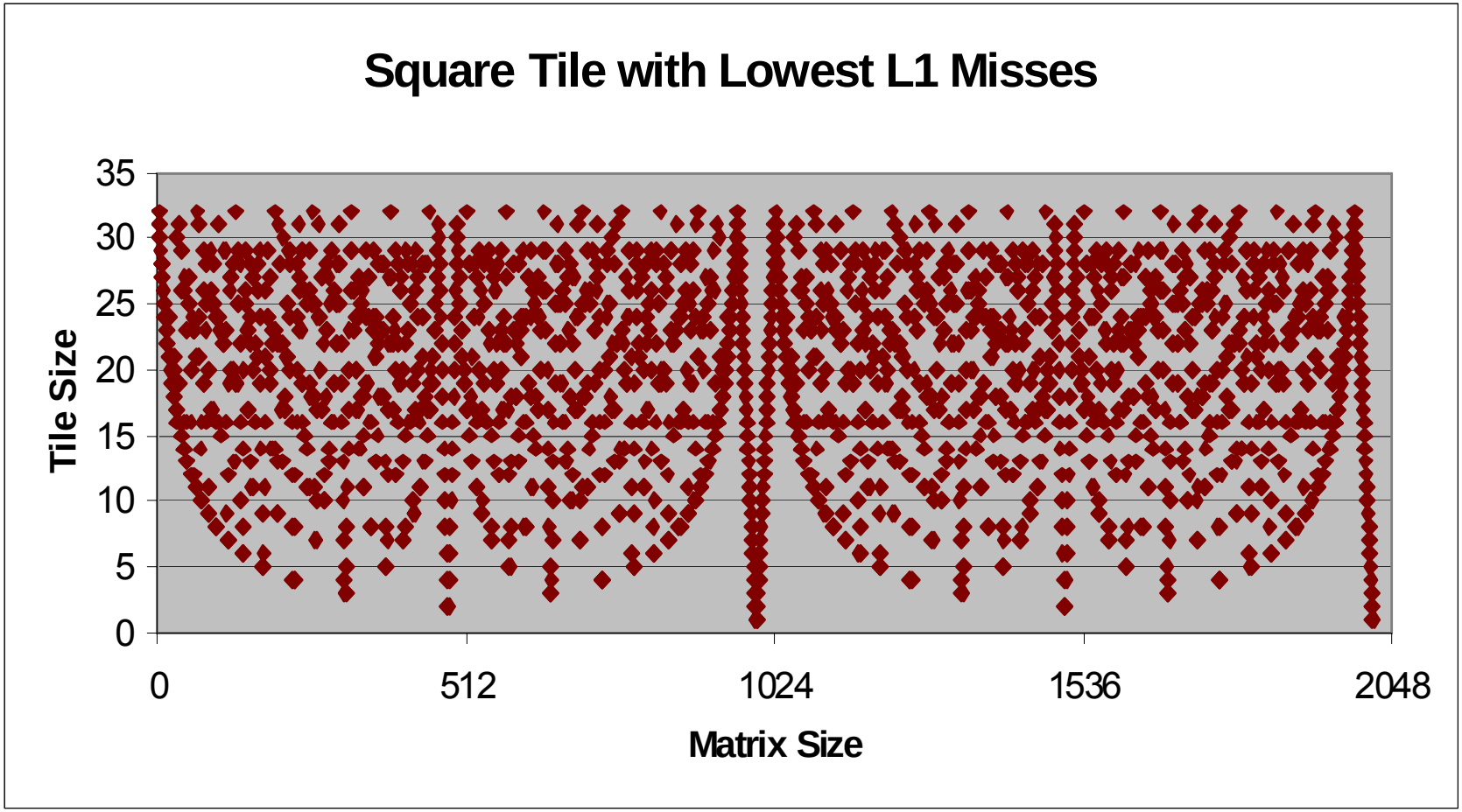
$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ -1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i1 \\ i2 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 100 \\ -1 \\ 1 \\ 100 \end{bmatrix}$$

Solution exist?
Yes ➔dependence

How do we get locality (in caches)?

- Data locality:
  - data is reused and is present in cache
  - same data or same cache line
- Data *reuse:*
  - data used multiple times
  - intrinsic in computation
- If a computation has reuse, what can we do to get locality?
  - code reordering transformations (today)
  - data layout

How to select optimal tile size ? (topic for next week)



Slide source: Jacqueline Chame

I. Motivation
Tiling for Real Caches

- Tiling reduces *capacity* misses
- In real life:
  - Caches are direct-mapped/small associativity
  - Tiling may introduce conflict misses:
    - Data within a tile is not contiguous in memory
  - Conflict misses may offset benefits of tiling
  - Tiling may conflict with other performance optimizations (*e.g.*, prefetching)

Overheads and Other Complexity

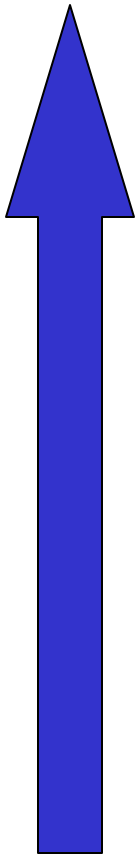
- Complex interactions lead to unpredictability
  - SIMD execution in SSE-3
  - Impact on hardware prefetching
  - Register spill
  - Interaction with instruction-level parallelism
  - Overhead of additional control for small problem sizes
  - …

II. The self-tuning Library ATLAS

- High-performance Basic Linear Algebra Subprograms (BLAS) library code
- Level-3 BLAS

  C = α A * B + βC

  - **A, B, C: matrices**
  - **α, β: scalars**
- Matrix multiplication: **α=1, β=0**
- Level-2 BLAS
  - matrix-vector operations
- Level-1 BLAS
  - vector-vector operations



Increased data reuse.

Increased performance gap between hand-tuned and compiled.

ATLAS Approach to Generating Optimized BLAS (GEMM)

- Sequence of experiments to apply code transformations on the high-level code
  - L1 cache-level tiling
  - register-level "tiling" (unroll-and-jam and scalar replacement)
  - scheduling computation and memory accesses in the inner loop body of the transformed loop nest (e.g., MAC instruction? SSE?)

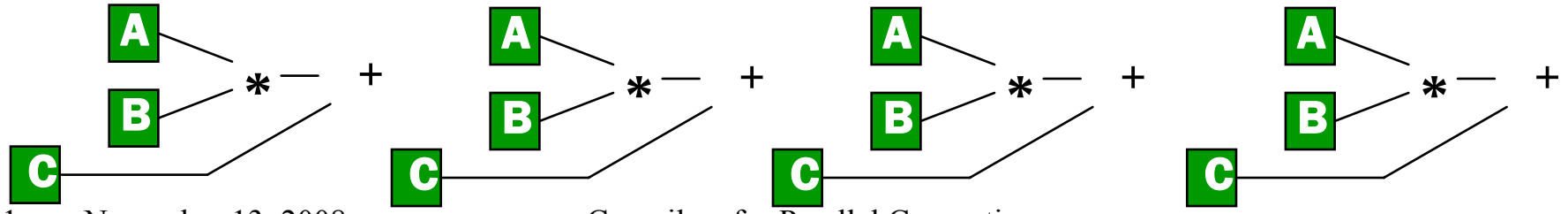
"Self-Adapting Linear Algebra Algorithms and Software", by J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley and K. Yelick. *Proceedings of the IEEE*, 93(2):293-312, February, 2005.

Aside: Unroll & Jam (Scalar Replacement on Next Slide)

UR&J equivalent to tiling followed by unroll inner tile (safe if tiling safe)

```
DO K = 1, N  by T_K
    DO I = 1, N by 4
        DO J = 1, N
            DO KK = K, min(KK+ T_K, N)
                C(I, J) = C(I, J) + A(I, KK) * B(KK, J)
                C(I+1, J) = C(I+1, J) + A(I+1, KK) * B(KK, J)
                C(I+2, J) = C(I+2, J) + A(I+2, KK) * B(KK, J)
                C(I+3, J) = C(I+3, J) + A(I+3, KK) * B(KK, J)
```

Now fine-grain parallel computations are exposed



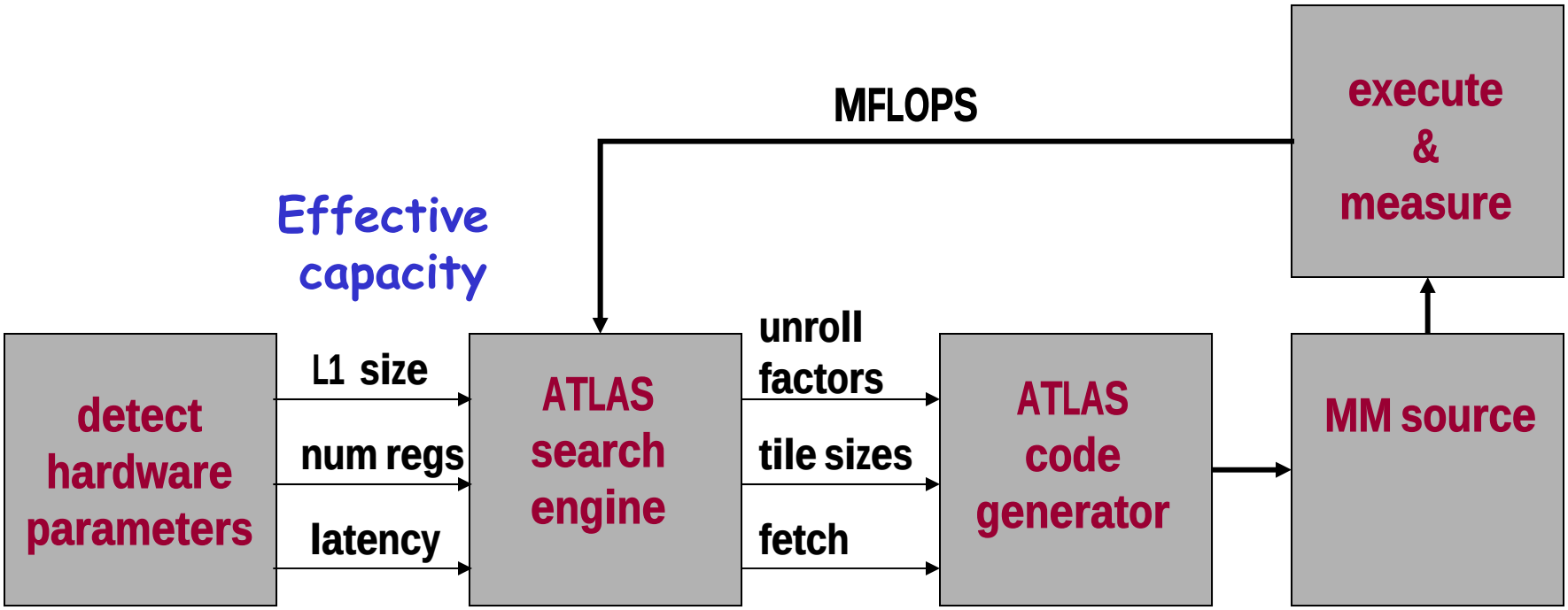
Aside: Scalar Replacement

Scalar Replacement: Replace accesses to C with scalars

```
DO K = 1, N  by T_K
    DO I = 1, N by 4
        DO J = 1, N
            C1 = C(I,J); C2 = C(I+1,J); C3 = C(I+2,J); C4 = C(I+3,J)
            DO KK = K, min(KK+ T_K, N)
                C1 = C1 + A(I, KK) * B(KK, J)
                C2 = C2 + A(I+1, KK) * B(KK, J)
                C3 = C3 + A(I+2, KK) * B(KK, J)
                C4 = C4 + A(I+3, KK) * B(KK, J)
            C(I,J) = C1; C(I+1,J) = C2; C(I+2,J)=C3; C(I+3,J) = C4
```

Now C accesses can be mapped to "named registers"

ATLAS empirical-driven optimizer



ATLAS Empirical Optimization (now called Autotuning)

- ATLAS search engine
  - performs empirical search to determine optimization parameter values
- ATLAS code generator
  - generates code given parameter values determined by search engine

Cache-level tiling

- Matrix multiplication is converted to a sequence of smaller matrix multiplications with data sets that fit in cache
  - *Mini-MMMs*
    - **MBxKB** sub-matrix of **A**
    - **KBxNB** sub-matrix of **B**
    - **MBxNB** sub-matrix of **C**
- Mini-MMM's also tiled to exploit data reuse in registers
  - *Micro-MMMs*
    - **MUx1** sub-matrix of **A**
    - **1xNU** sub-matrix of **B**
    - **MUxNU** sub-matrix of **C**

ATLAS mini-MMMs

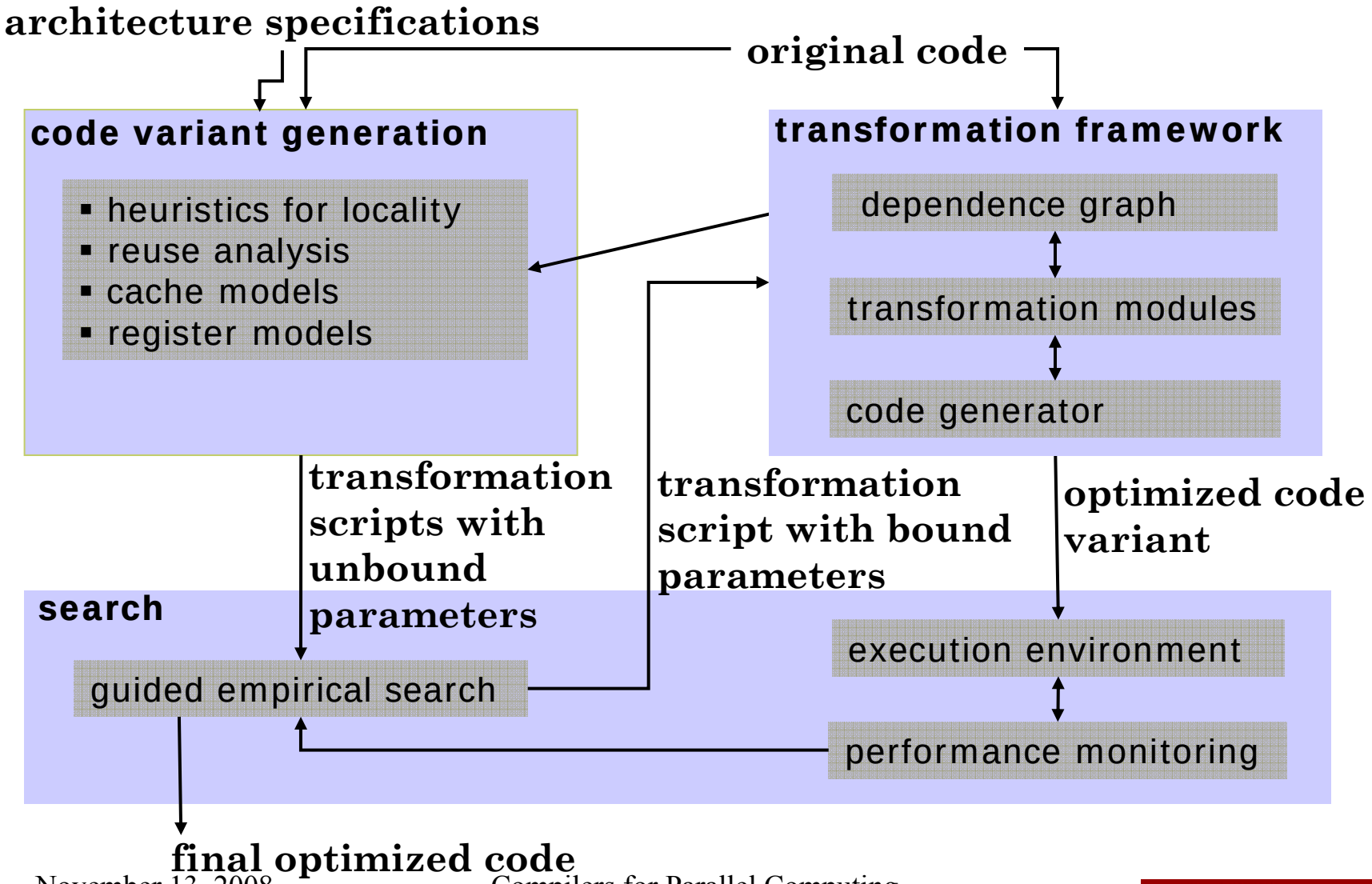
- Tiling for L1 cache only
- Square tiles only: **NB = MB = KB**
- **NB** is an optimization parameter

```
/* mini-MMM of size NBxNB */
for (j = 0; j < NB; j ++)
    for (i = 0; i < NB; i ++)
        for (k = 0; k < NB; k ++)
            C[i][j] += A[i][k] * B[k][j];
```

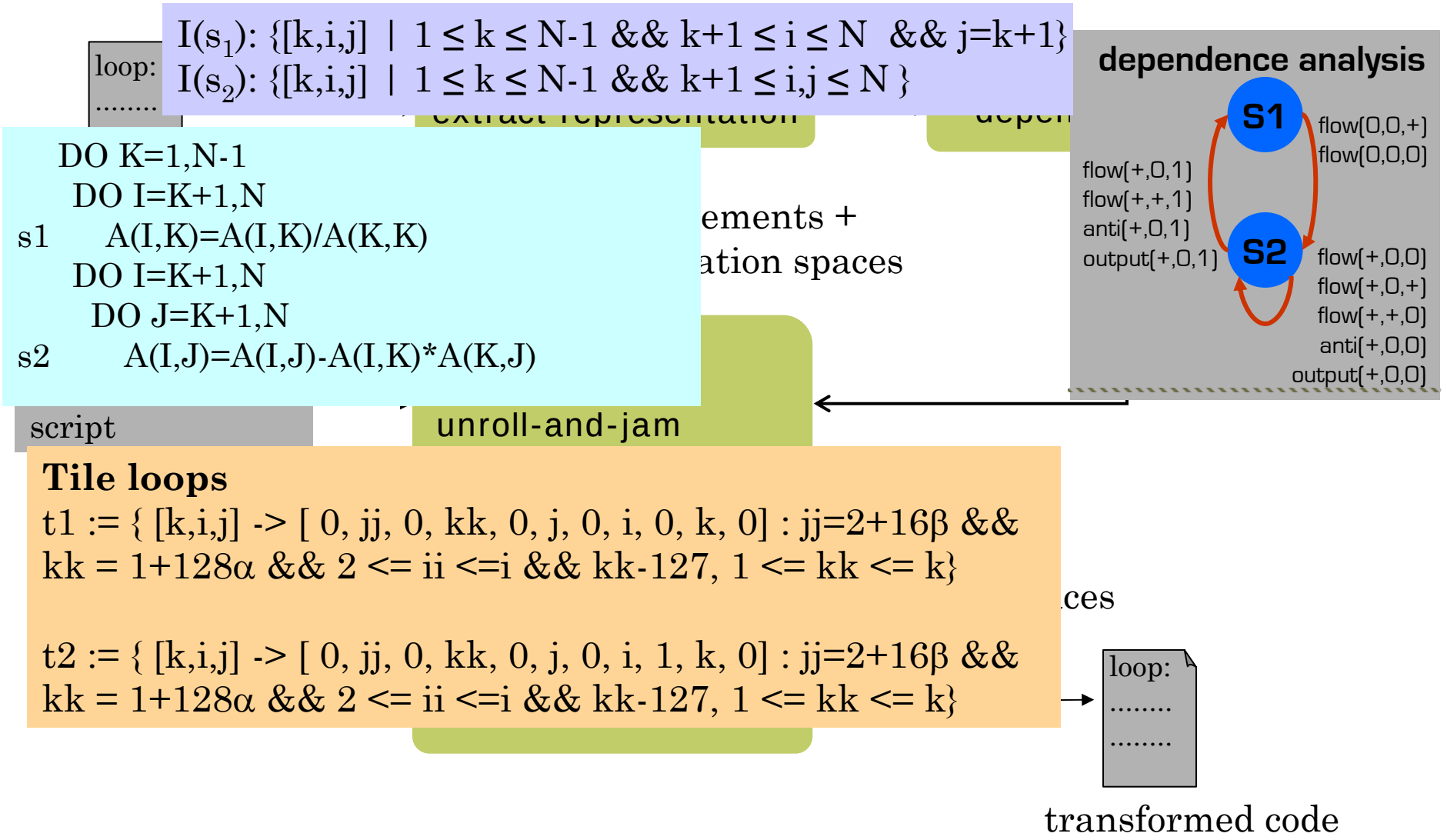
III. How to Generalize in a Compiler?

- Imperfect loop nests and a large suite of code transformations
- Automate code generation and empirical search
  - Can also automate derivation of optimization strategy
- Provide interface to sophisticated application developer
  - Provides high-level description of experiments to be run

TUNE Compiler: Model-Guided Empirical Optimization

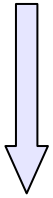


Polyhedral Transformation Framework



LU Example: Polyhedral Loop Transformation Framework

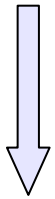
Existing iteration space:
is1: {[k,i,j] | 1≤k≤N-1 Λ k+1≤i≤N Λ j=k+1}
is2: {[k,i,j] | 1≤k≤N-1 Λ k+1≤i,j≤N }



Mapping relations:
t1: {[k,i,j]->[0,k,0,i,0,j,0]}
t2: {[k,i,j]->[0,k,0,i,1,j,0]}

auxiliary loops for lexicographical order of different loops at the same loop level

Transformed iteration space:
is1: {[0,k,0,i,0,j,0] | 1≤k≤N-1 Λ k+1≤i≤N Λ j=k+1}
is2: {[0,k,0,i,1,j,0] | 1≤k≤N-1 Λ k+1≤i,j≤N }



Omega code generation

```
DO T2=1,N-1
   DO T4=T2+1,N
      A(T4,T2)=A(T4,T2)/A(T2,T2)
       DO T6=T2+1,N
          A(T4,T6)=A(T4,T6)-A(T4,T2)*A(T2,T6)
```

Transformation Script for LU Factorization

```
   DO K=1,N-1
     DO I=K+1,N
s1     A(I,K)=A(I,K)/A(K,K)
     DO I=K+1,N
       DO J=K+1,N
s2       A(I,J)=A(I,J)-A(I,K)*A(K,J)
```

```
permute([0,1,2])
tile(s1,5,64,1)
split(s1,3,[d3 ≤ d1-2])
permute(s2,[1,3,7,5])
permute(s1,[1,5,7,3])
split(s1,3,[d3 ≥d1-1])
tile(s3,3,32,3)
split(s3,5,[d9 ≤ d3-1])
tile(s3,9,32,5)
datacopy(s3,7,2,1)
datacopy(s3,7,3)
unroll(s3,9,4)
tile(s1,7,32,3)
tile(s1,5,32,5)
datacopy(s1,7,2,1)
datacopy(s1,7,3)
unroll(s1,9,4)
```

TRSM

GEMM

separate perfect and imperfect loop nests

separate non-overlapping read and write accesses

Automatically Generated Code for LU

```
REAL*8 P1(32,32),P2(32,64),P3(32,32),P4(32,64)
OVER1=0
OVER2=0
DO T2=2,N,64
  IF (66<=T2)
   DO T4=2,T2-32,32
     DO T6=1,T4-1,32
       DO T8=T6,MIN(T4-1,T6+31)
         DO T10=T4,MIN(T2-2,T4+31)
           P1(T8-T6+1,T10-T4+1)=A(T10,T8)
       DO T8=T2,MIN(T2+63,N)
         DO T10=T6,MIN(T6+31,T4-1)
           P2(T10-T6+1,T8-T2+1)=A(T10,T8)
       DO T8=T4,MIN(T2-2,T4+31)
         OVER1=MOD(-1+N,4)
         DO T10=T2,MIN(N-OVER1,T2+60),4
           DO T12=T6,MIN(T6+31,T4-1)
             A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
             A(T8,T10+1)=A(T8,T10+1)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+1-T2+1)
             A(T8,T10+2)=A(T8,T10+2)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+2-T2+1)
             A(T8,T10+3)=A(T8,T10+3)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+3-T2+1)
         DO T10=MAX(N-OVER1+1,T2),MIN(T2+63,N)
           DO T12=T6,MIN(T4-1,T6+31)
             A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
     DO T6=T4+1,MIN(T4+31,T2-2)
       DO T8=T2,MIN(N,T2+63)
         DO T10=T4,T6-1
           A(T6,T8)=A(T6,T8)-A(T6,T10)*A(T10,T8)
```

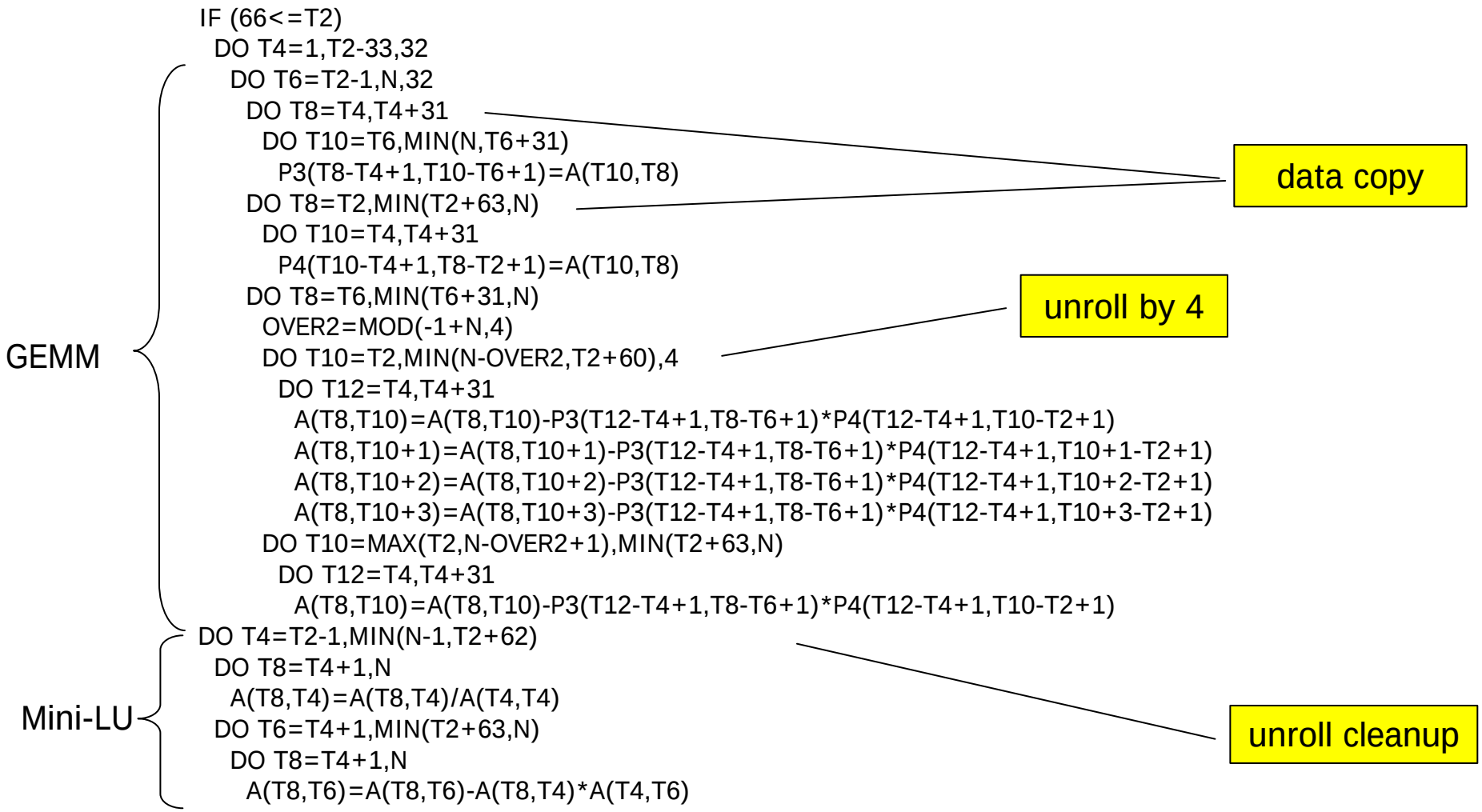
TRSM

data copy

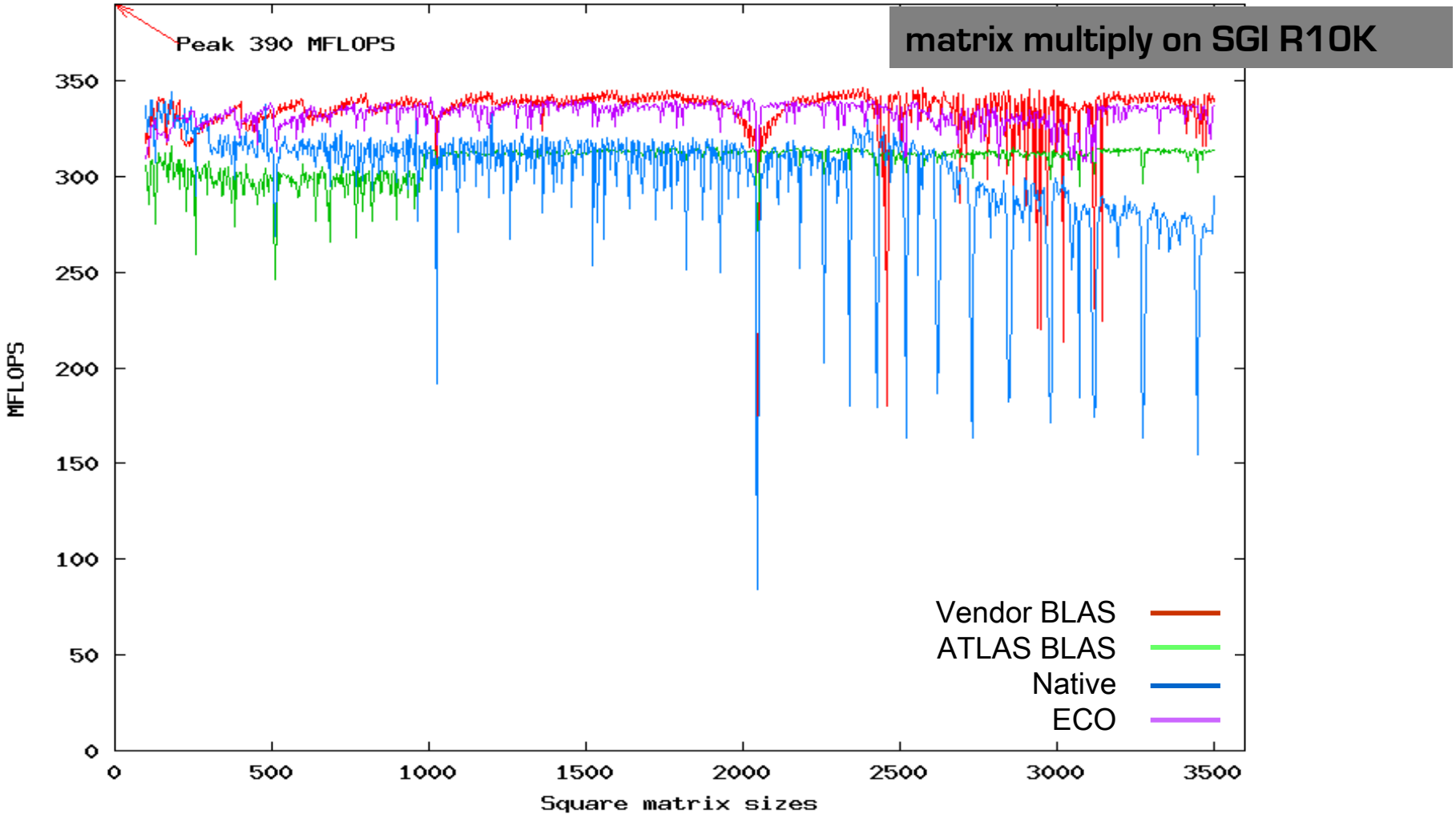
unroll by 4

unroll cleanup

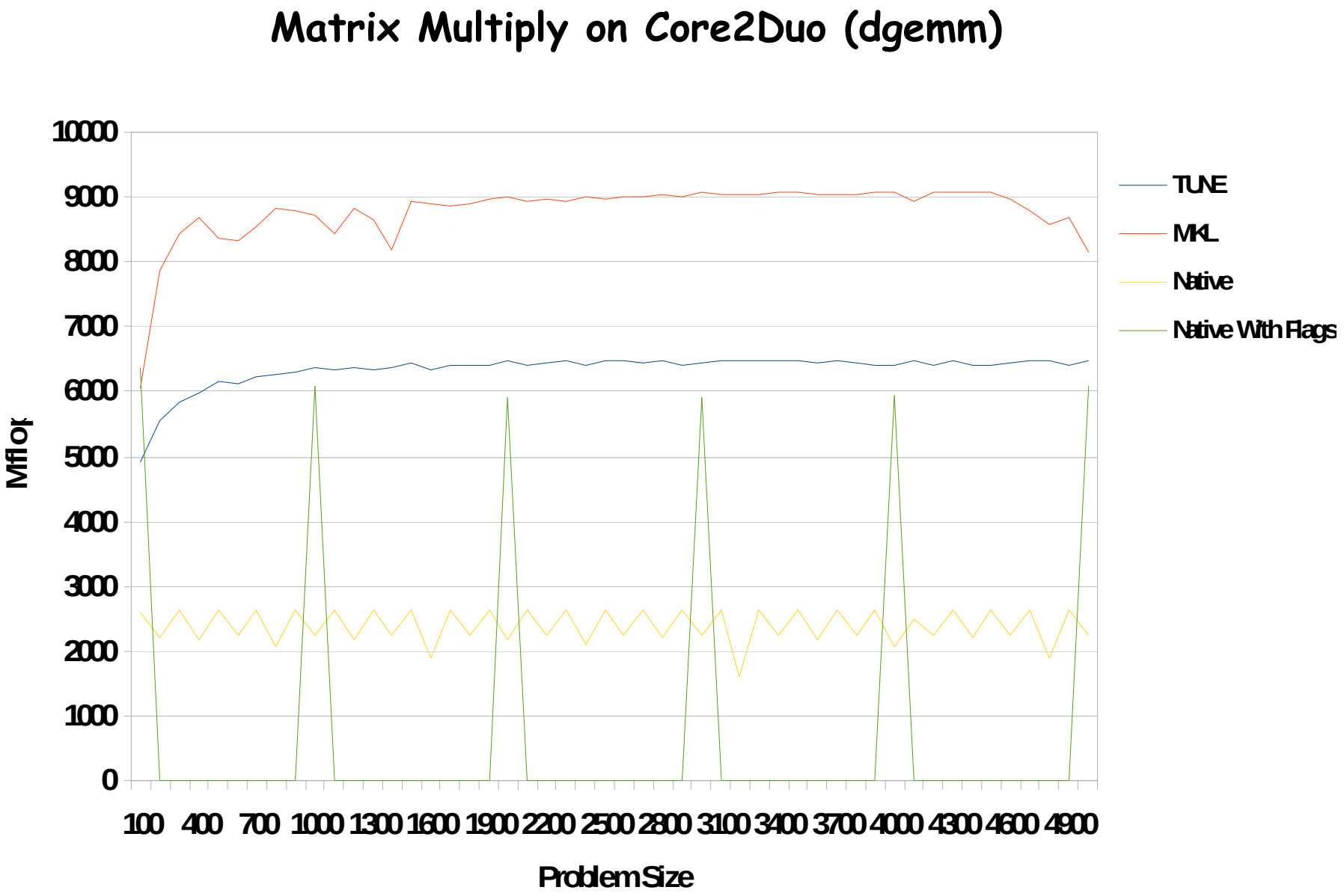
Automatically-Generated Code for LU (Cont.)



Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler



Preliminary Core2Duo Results



Differences: PAPI Measurements

	MKL	TUNE
SSE_PrefNta_Ret	126362	0
SSE_PrefT1_Ret	32260262	0
SSE_PrefT2_Ret	0	0
SSE_PrefNta_Miss	46467	0
SSE_PrefT1_Miss	1038617	0
SSE_PrefT2_Miss	0	0
DCache_Rep	332297749	18360367
DCache_Pend_Miss	39019994	140968429
Data_Mem_Ref	417906963	578392107
Pref_Rqsts_Up	54180035	30368079
Pref_Rqsts_Dn	1649884	14441
UnhltCore_Cycles	545770204	761735797

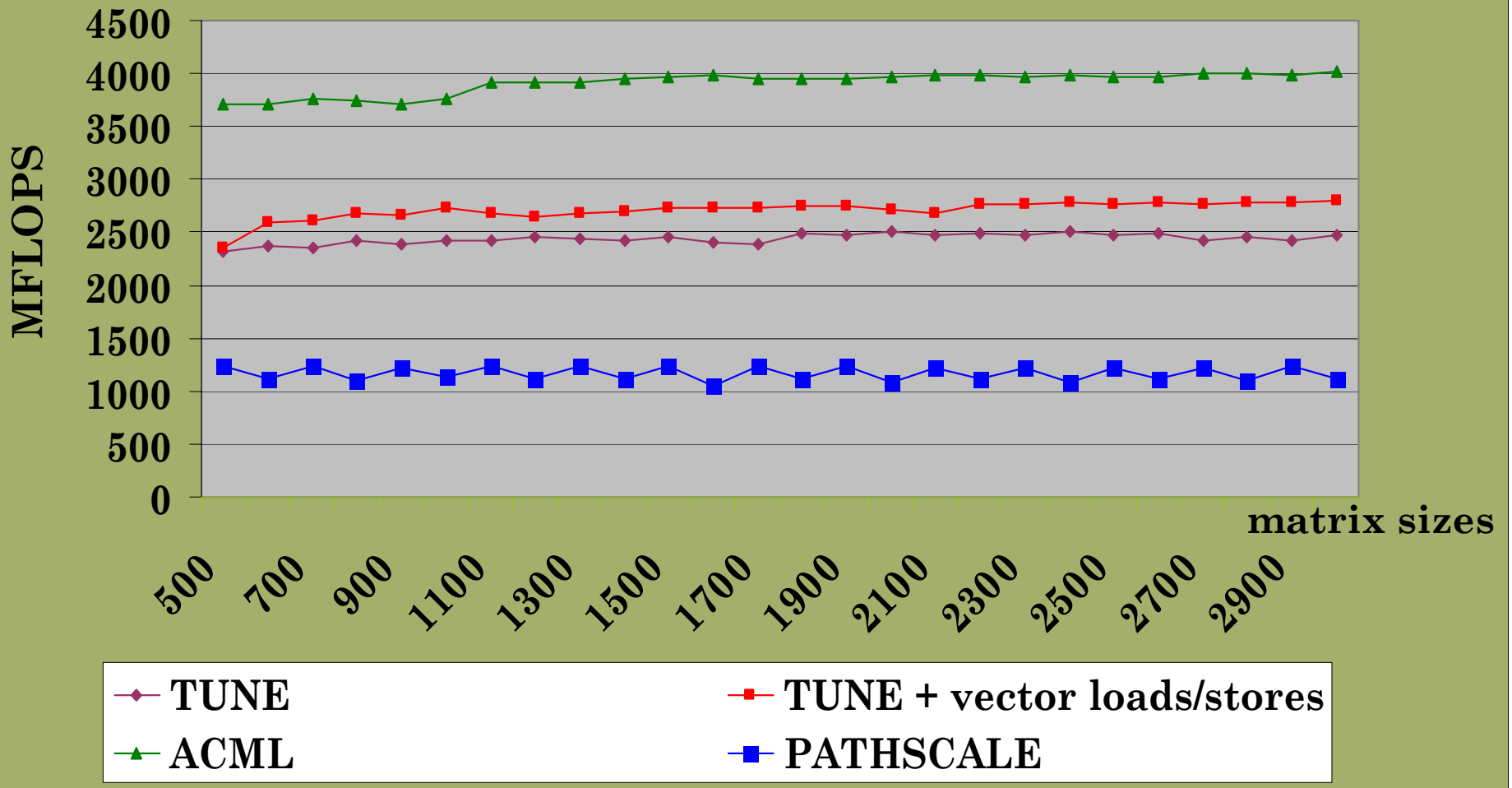
Need Software Prefetching

More L1 Misses Lower Penalty

Fewer References Hardware Prefetching

28% gap

Matrix Multiply on Jacquard (Opterons at NERSC)



TUNE: TUNE for locality, PATHSCALE for vectorization
ACML: hand-tuned vendor library
PATHSCALE: not vectorized (alignment issues)

Original Code Variant Generation Algorithm

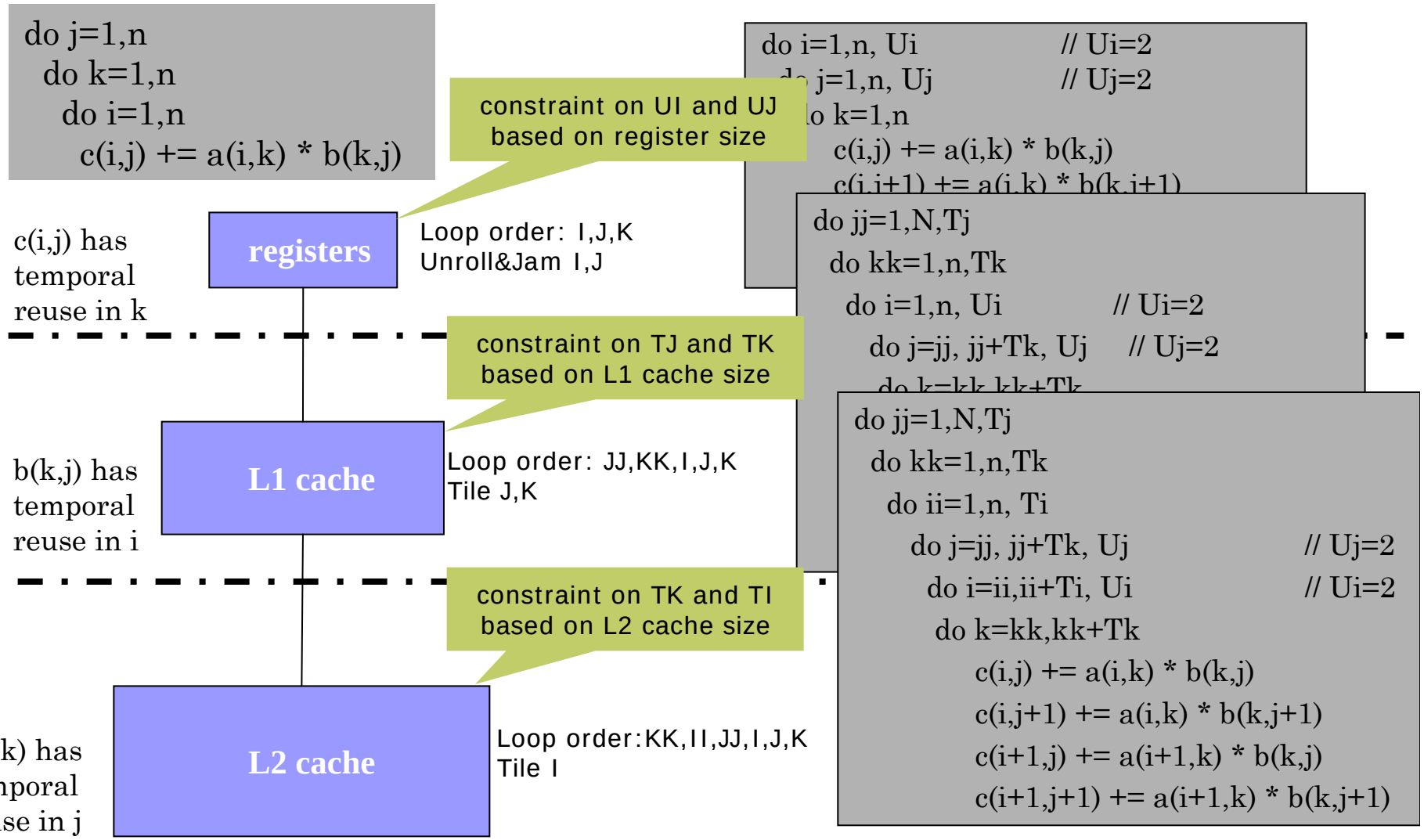
- **Key Insights:**
  - Target data structures to specific levels of the memory hierarchy based on reuse analysis
  - Compose code transformations and determine constraints

For each memory hierarchy level in (Register, L1, L2, ...), ***use models*** to:

1. Select the data structure *D* which has maximum reuse from reuse analysis (if possible, one that has not been considered)
2. Permute the relevant loops and apply tiling (unroll-and-jam for registers) according to newly selected reuse dimension
3. Generate copy variant if copying is beneficial
4. Determine constraints based on *D* and current memory hierarchy level characteristics, using register/cache/TLB footprint analysis
5. Mark *D* as considered

[CGO05]

Mapping Reuse to Memory Levels



Example: Transformation Scripts for Matrix Multiply

code variant I

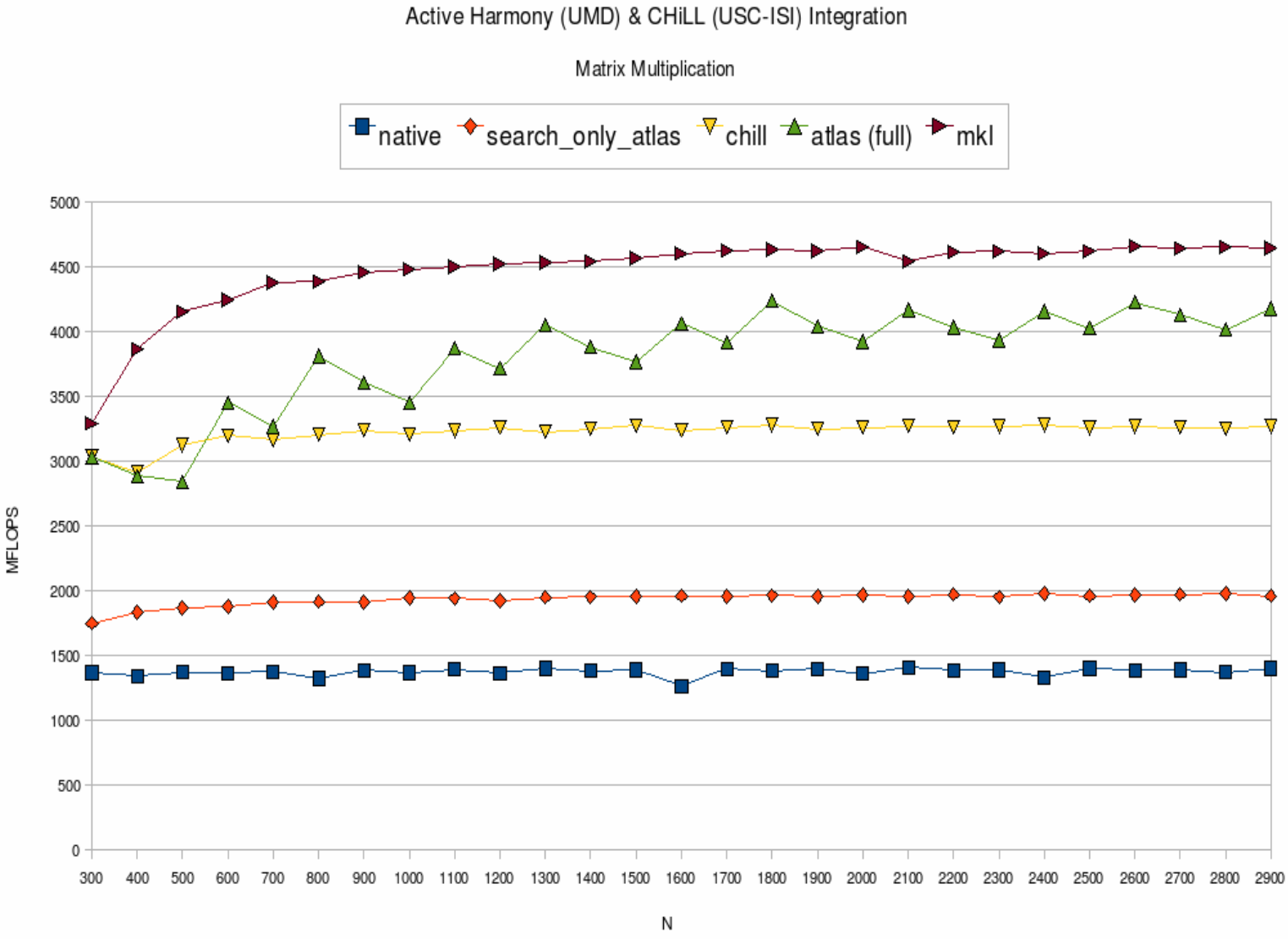
```
permute(2,0,1)
tile(s0,3,Tj)
tile(s0,3,Ti)
tile(s0,9,Tk)
datacopy(s0,5,2,1)
datacopy(s0,7,3,0)
unroll (s0,7,Ui)
unroll (s0,9,Uj)
```

code variant II

```
permute(s0,2,1)
tile(s0,3,Ti)
tile(s0,3,Tj)
tile(s0,9,Tk)
datacopy(s0,5,3)
datacopy(s0,7,2,
        transposed)
unroll (s0,7,Uj)
unroll (s0,9,Ui)
```

Ti, Tj, Tk, Ui, Uj are ***unbound parameters***

Integration with Parameter Search Framework (Parallel Simplex)



Tiling and Parallelization

- Parallel machines
  - Conflicting goals if
    - parallel loop also carries reuse
  - Works well if
    - coarse-grain parallelism (outer loops)
    - inner loops carry reuse
  - Data partitioning may cause false sharing
    - choose block sizes multiple of cache line size

Impact of SSE and other Multimedia Extensions

- Data must be contiguous in memory for SIMD operations (spatial reuse)
  - Transpose during data copy accomplishes this for GEMM
- Data alignment
- Small number of SSE registers changes register locality strategy
- Native compilers are limited in generating high-quality code
- Tradeoffs for ILP, loop-level parallelism and SIMD parallelism

Future Research Directions

- Much work to be done on managing large optimization search spaces
  - A combination of models, pruning heuristics, search algorithms
  - Optimization search spaces grow as complexity increases: composing decisions, increased architectural complexity, more global optimizations
- New architectures
  - Multi-core with complex memory hierarchies (e.g., NUCA)
  - Specialized processors such as GPUs (CS6963!)
  - Heterogeneous platforms (partitioning, code generation, composition)

Future Research Challenges, cont.

- Effective interactions with application programmers
  - Understanding what is useful by working on real applications
  - New transformations (e.g., data reorganization)
  - Mechanisms for code generation and auto-tuning integrated into code
- Validation
  - Does a transformation script represent a valid reordering of the original sequential computation?