



Information Sciences Institute

# Compiler Autotuning for Locality Optimization

Chun Chen, Jacqueline Chame,  
Muhammad Murtaza, Mary Hall



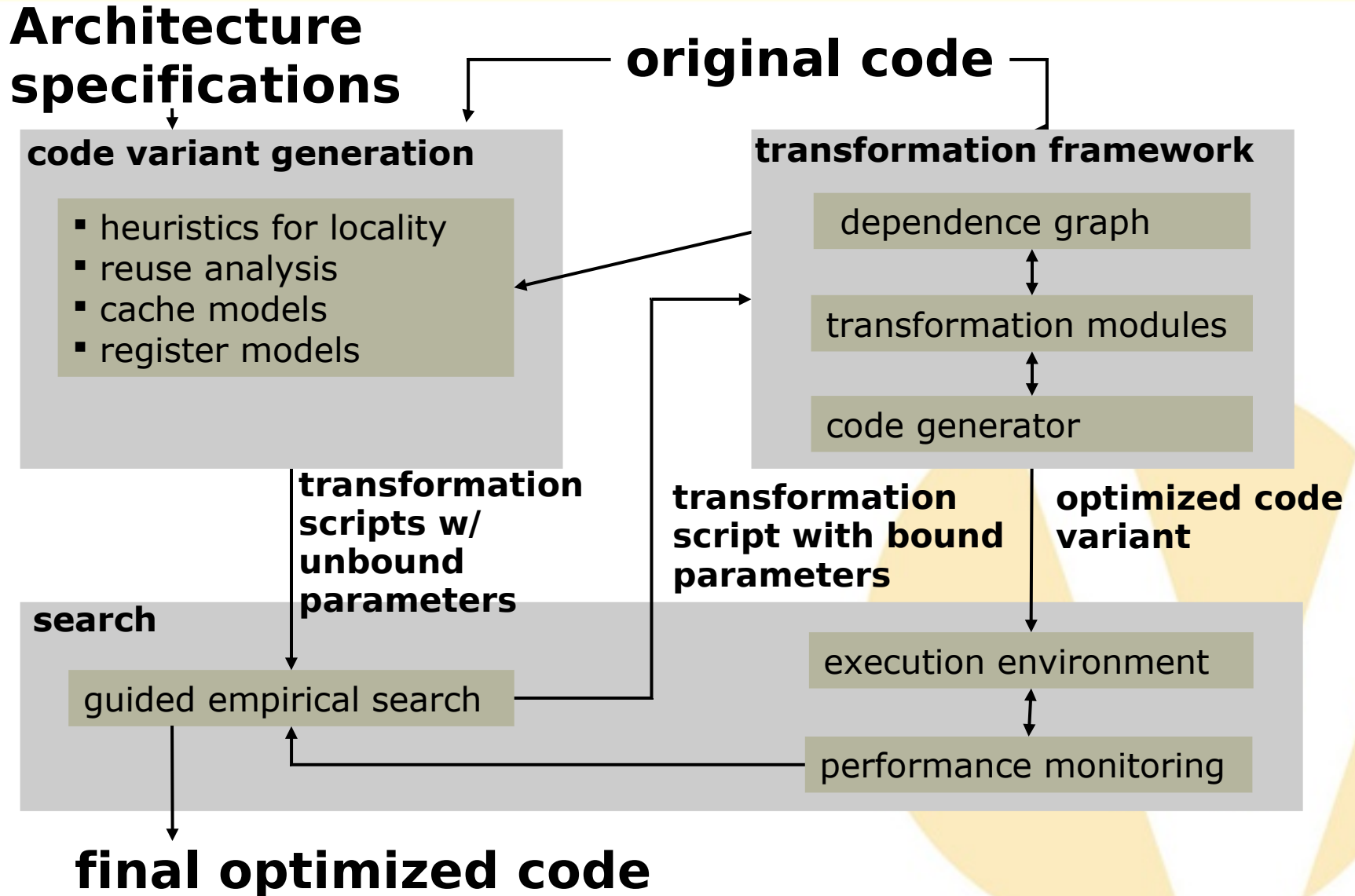
**USC Viterbi**  
School of Engineering

## Recap of Existing Compiler Limitations

- Data reuse mostly focuses on single cache level or idealistic cache model
  - Miss the opportunity of efficient data reuse across the entire memory hierarchy.
- Lack of mechanism to compose complex high-level transformations
  - Built-in rigid transformation strategy often generates very different code from best known manually-optimized, with relatively low performance.

- Data reuse strategy
  - Reuse different data structure in different memory hierarchy level.
  - At each level, the targeted reuse data can be copied if the amount of reuse permits. Other data appears to be streaming at this level.
- Composing high-level loop transformation
  - Manipulate complex loop nest in iteration spaces completely (including imperfect loop nest).
  - Complex loop transformation is a sequence of mapping of iteration spaces.

# Model-Guided Empirical Optimization



# Simplified Outline of Code Variant Generation Algorithm

$U$  := set of uniformly generated sets

$M$  := set of memory hierarchy levels {Register, L1, L2...}

**For** each uniformly generated set(s) in  $U$  in a priority order decided by reuse analysis, delete them from  $U$

**If** dim. of temporal reuse space equals to dim. of the iteration space

Permute loops according to array layout if it is legal

**If** there is temporal reuse or group reuse

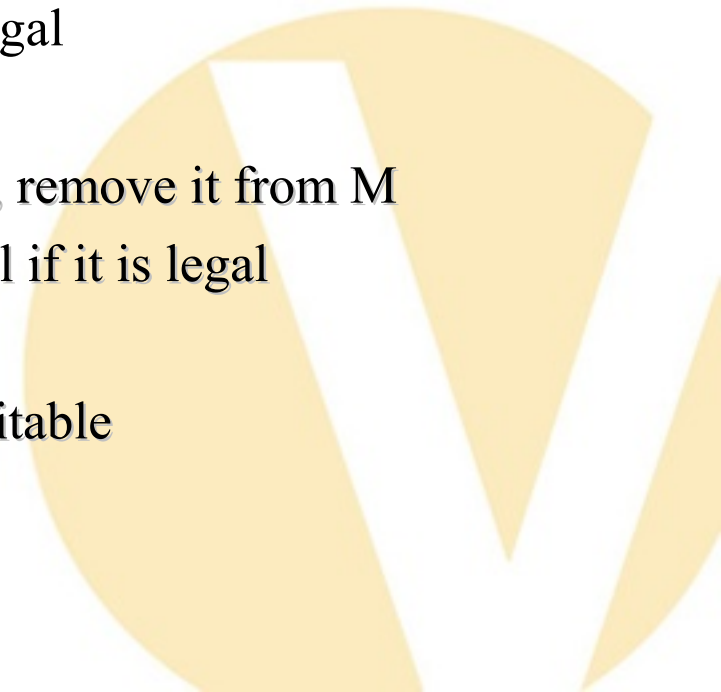
Pick the fastest memory hierarchy level from  $M$ , remove it from  $M$

Tile the loop targeting the selected memory level if it is legal

Determine constraints on parameters

Apply data copy if it is temporal reuse and profitable

Add prefetching within the innermost loops



# Code Variants Represented As Transformation Scripts

- Script: sequence of transformations
  - `permute([stmt], order)`
  - `tile(stmt, loop, tilesize, [outerloop])`
  - `unroll-and-jam(stmt, loop, unrollsize)`
  - `datacopy(stmt, loop, array)`
  - `split(stmt, loop, condition)`
  - ...

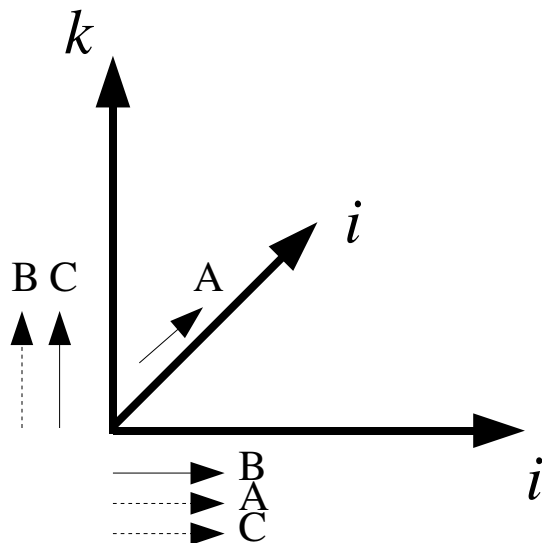


- Matrix Multiply
  - Multiple data structures
- Matrix-Vector Multiply
  - Dominant data structure
- LU Factorization
  - Splitting decision

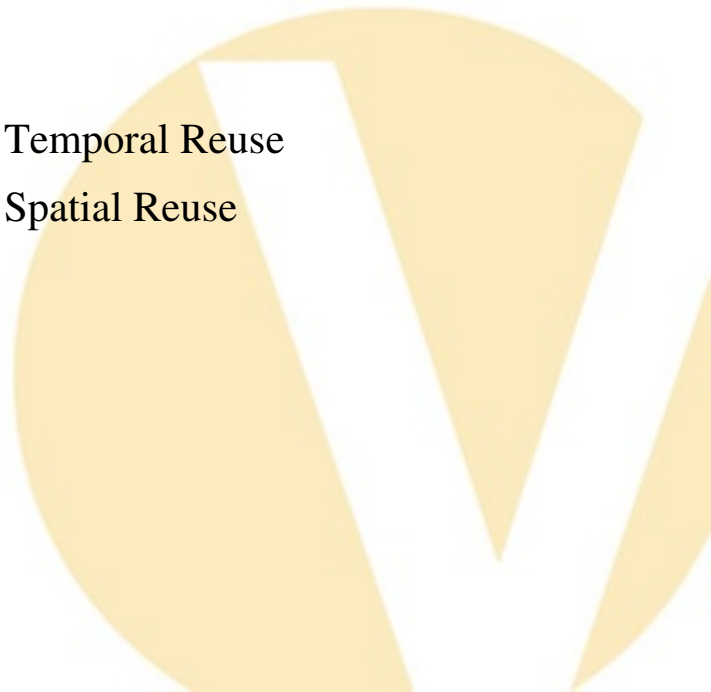


# Matrix Multiply Example

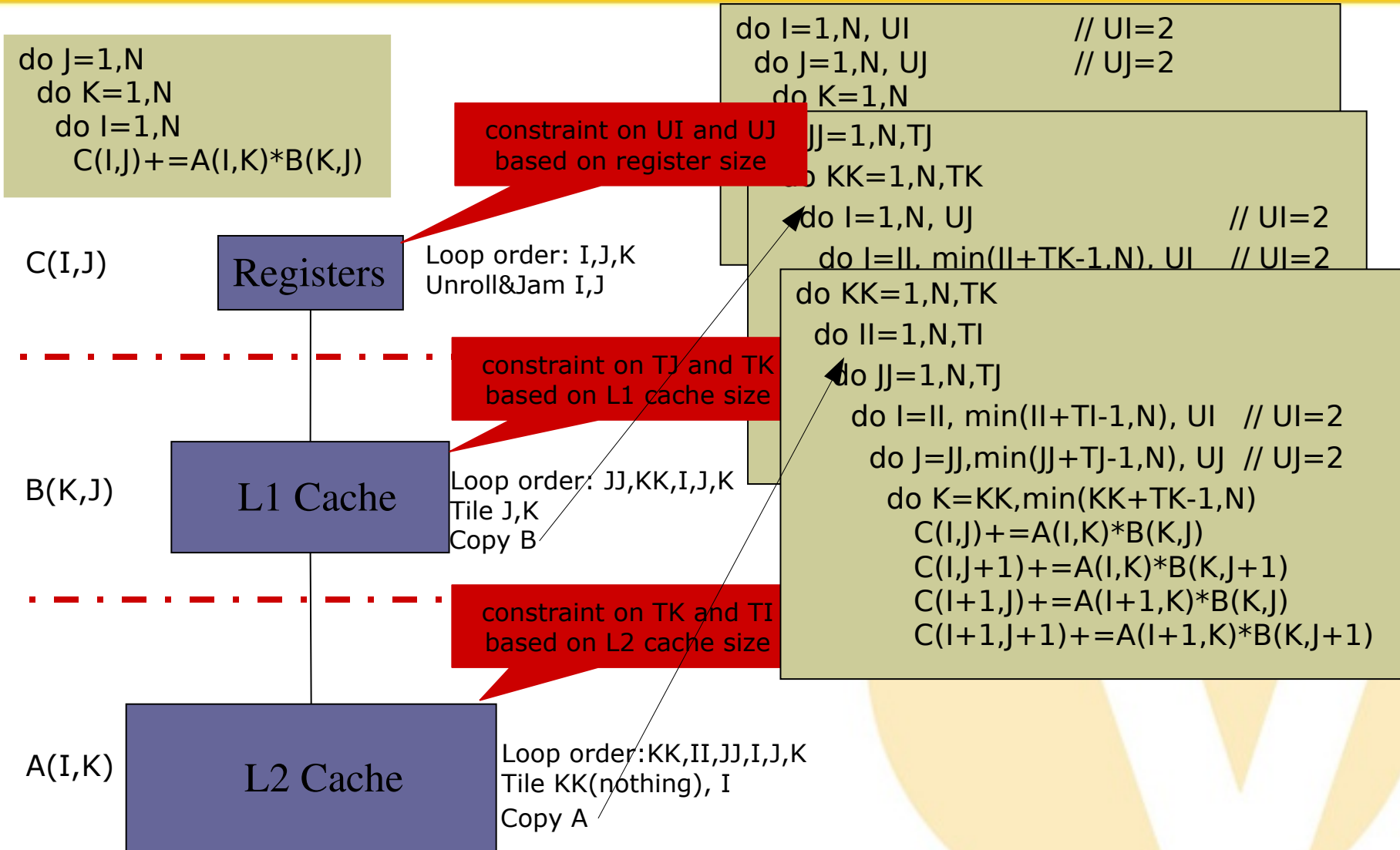
```
DO I=1,N
  DO K=1,N
    DO J=1,N
      C(I,J)+=A(I,K)*B(K,J)
```



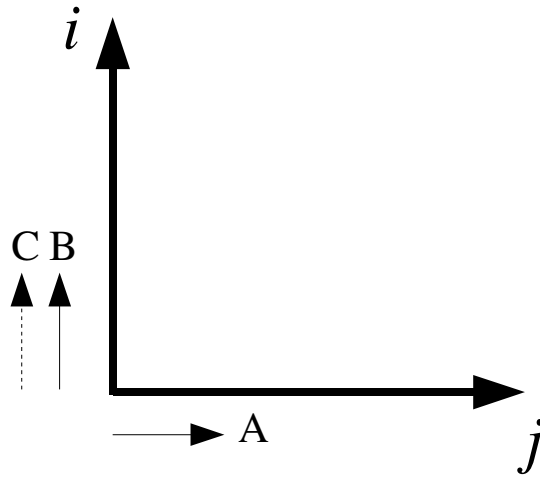
—▶ Temporal Reuse  
 - - -▶ Spatial Reuse



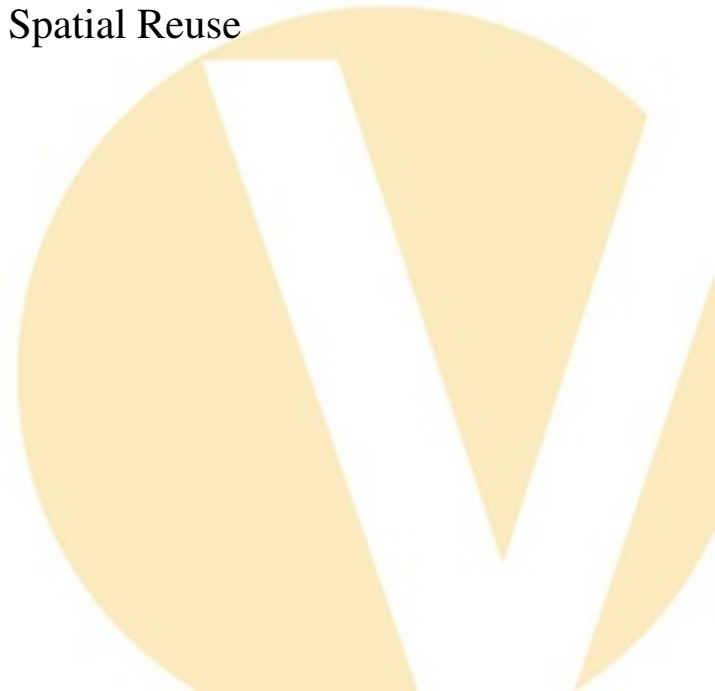
# Mapping reuse to memory levels



```
DO I=1,N  
  DO J=1,N  
    A(I)=A(I)+C(I,J)*B(J)
```



—▶ Temporal Reuse  
- - -▶ Spatial Reuse



```
do I=1,N
  do J=1,N
    A(I)+=C(I,J)*B(J)
```

## large array size:

1. C(I,J): streaming for cache

Loop order: J,I

2. reuse B(J) on I for registers

Unroll J

3. prefetch C

## small array size (2nd variant):

1. reuse A(I) on J for registers

Loop order: I,J

Unroll I

2. prefetch C

```
do J=1,N
  do I=1,N
    do J=1,N,UJ //UJ=2
      do I=1,N
        A(I)+=C(I,J)*B(J)
        A(I)+=C(I,J+1)*B(J+1)
```

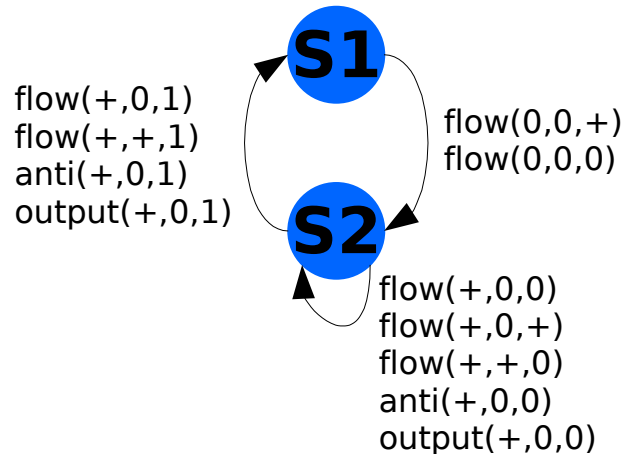
```
do I=1,N,UI //UI=2
  do J=1,N
    A(I)+=C(I,J)*B(J)
    A(I+1)+=C(I+1,J)*B(J)
```

# 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,K)*A(K,J)

```





```

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

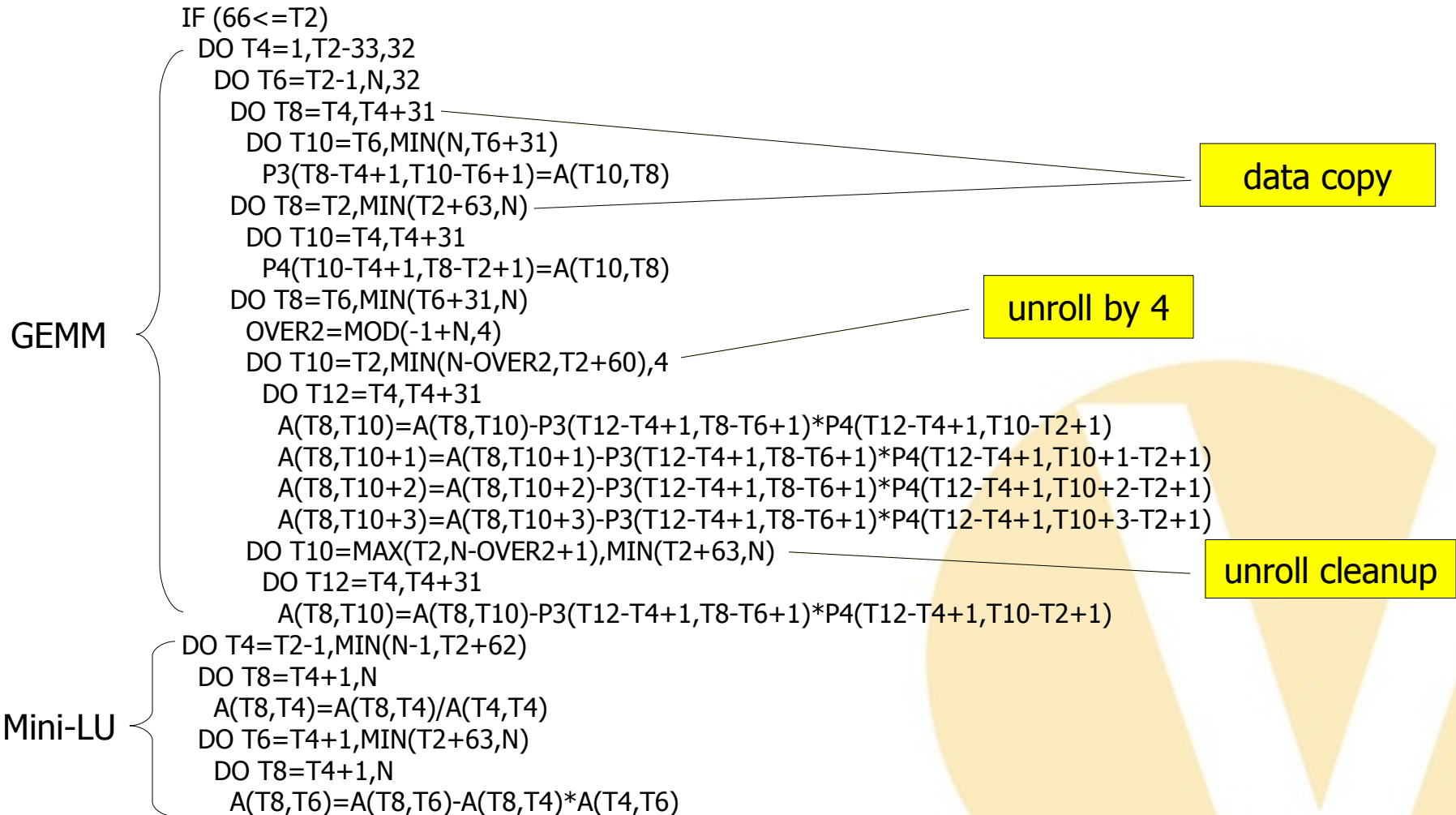
data copy

unroll by 4

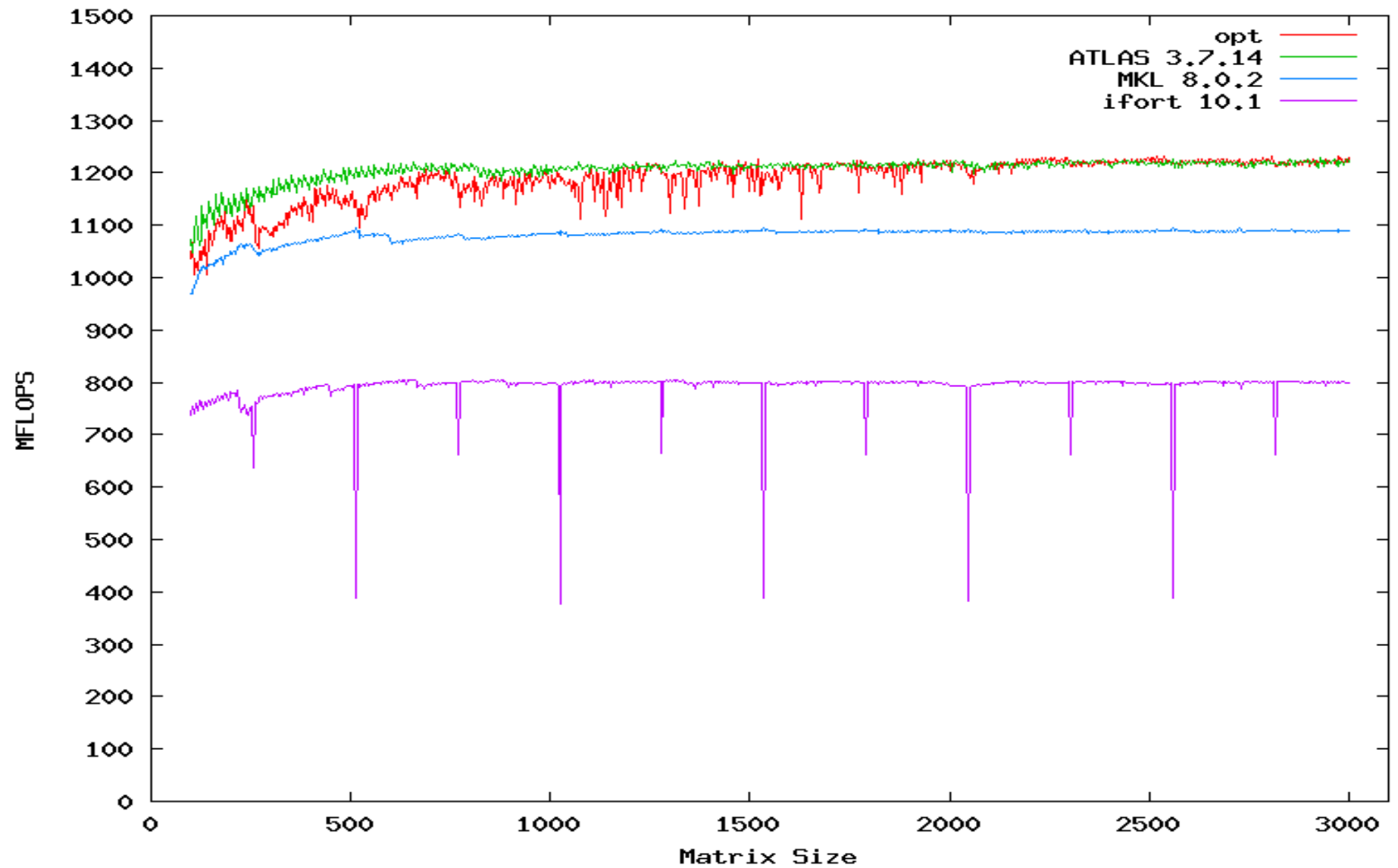
unroll cleanup



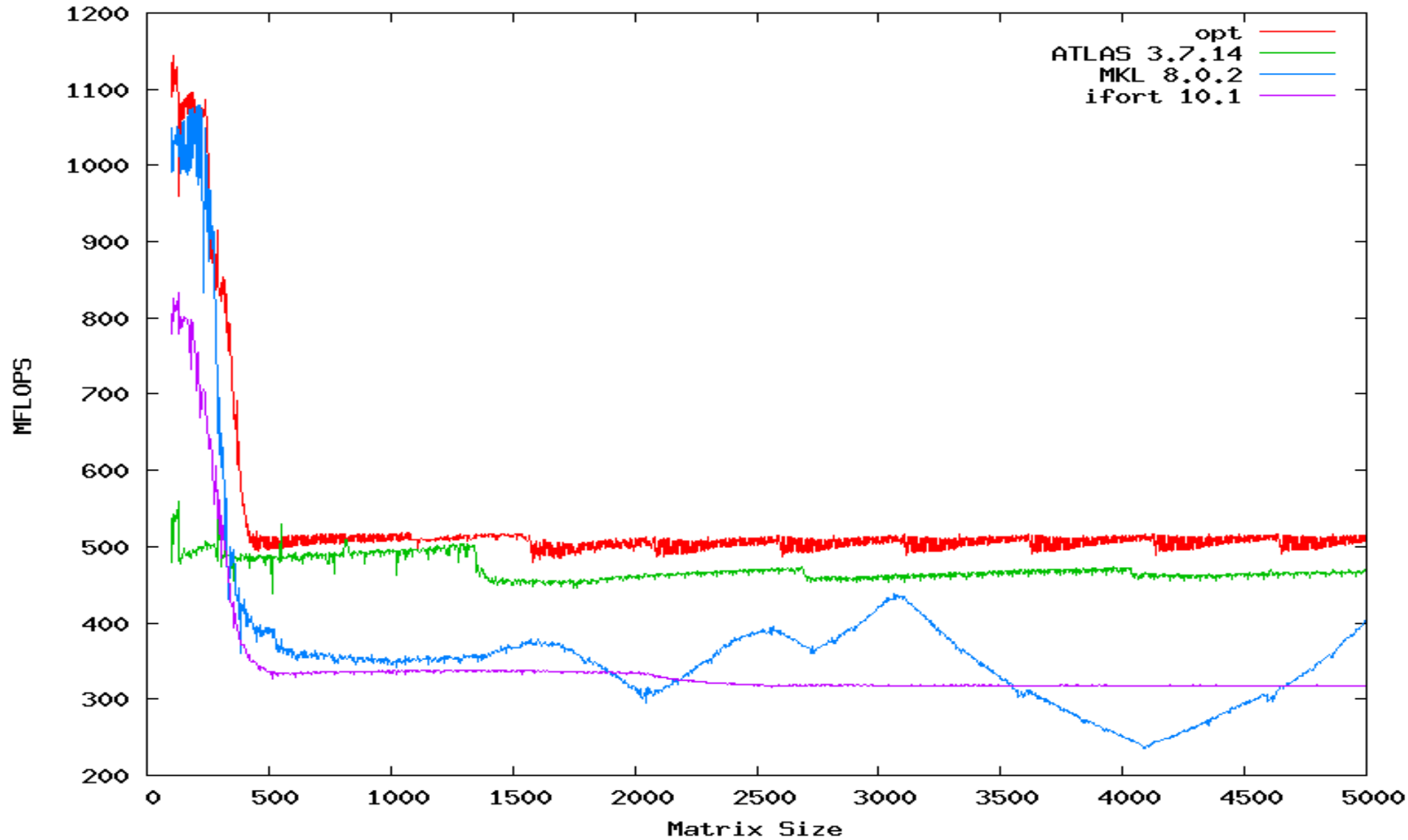
# Transformed Code for LU (Cont.)



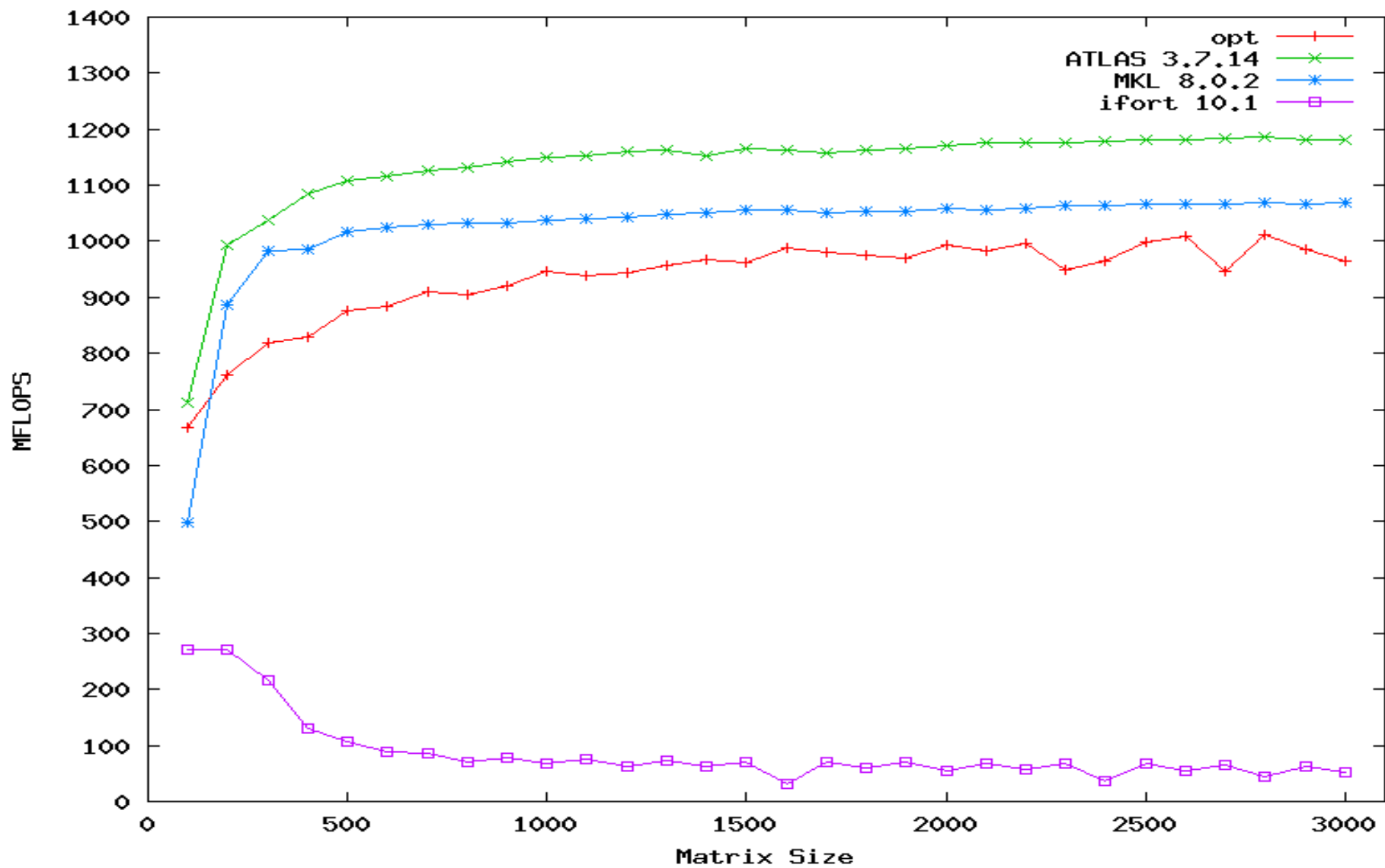
# Matrix Multiply on Pentium M



# Matrix-Vector Multiply on Pentium M



# LU on Pentium M



- Systematic approach to automatic performance tuning
- Robust transformation framework, convenient to search, can be integrated into other compilers
- Can achieve performance close to hand-tuned with compiler approach



- Explicit data management for new architectures like GPU and Cell, instead of relying on implicit cache behavior
- Fully integrate data copy decision with SIMD/vector optimization in addition to data reuse analysis
- More user-friendly interface to transformation script like referring loops with loop indices (for manual tuning)

Question?

