

Course Outline

1. Structure of a Compiler

2. Instruction Level Parallelism Optimizations (Josep Llosa)

- Instruction Level Parallelism
- Machine Independent Optimizations
- Instruction Scheduling
- Register Allocation

3. Memory Hierarchy Optimizations (J.R. Herrero)

- **Basic Concepts** Acknowledgement: Marta Jiménez
- **Basic transformations**
- **Loop vectorization**

4. Thread Level Parallelism Optimizations (Marc González)

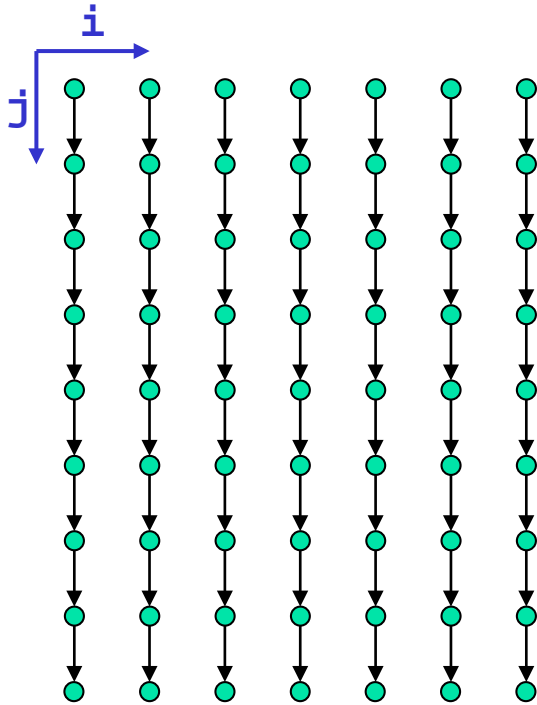
- Thread level Parallelism
- Analysis and detection of parallelism
- Programming models
- Parallel execution
- Memory models

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

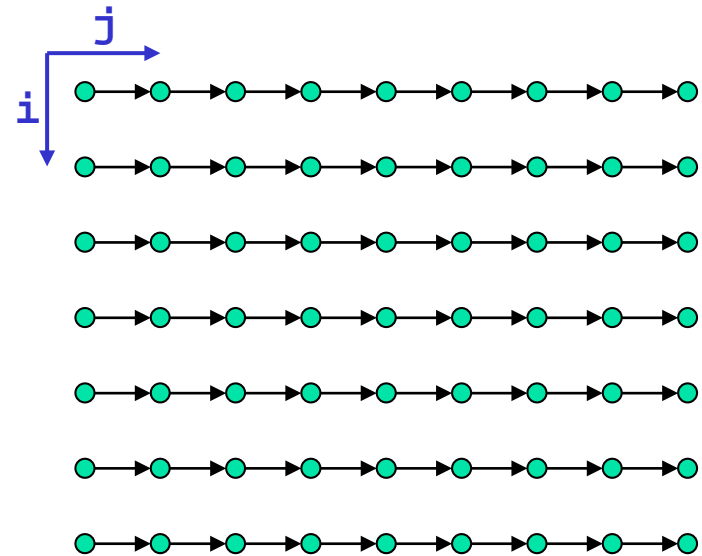
Loop Interchange

```
for (j=1; j<=M; j++)  
  for (i=1; i<=N; i++)  
    A(i,j) = F(A(i,j-1));
```



$$d_1 = (1, 0)$$

```
for (i=1; i<=N; i++)  
  for (j=1; j<=M; j++)  
    A(i,j) = F(A(i,j-1));
```

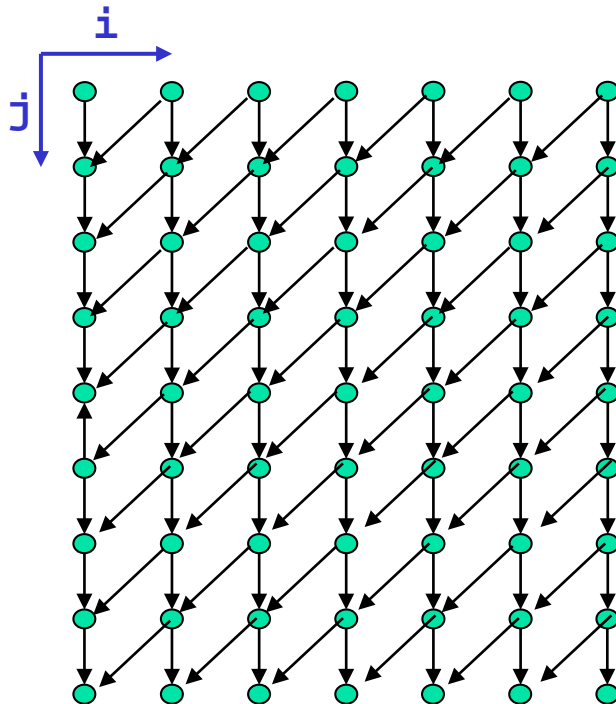


$$d'_1 = (0, 1)$$

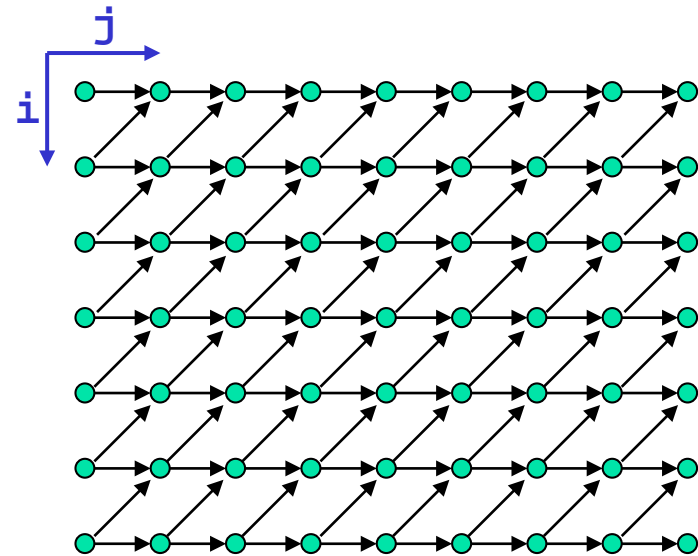
Loop Interchange

```
for (j=1; j<=M; j++)  
  for (i=1; i<=N; i++)  
    A[i,j] = F(A[i,j-1], A[i+1,j-1]);
```

Interchange is
not legal!!!



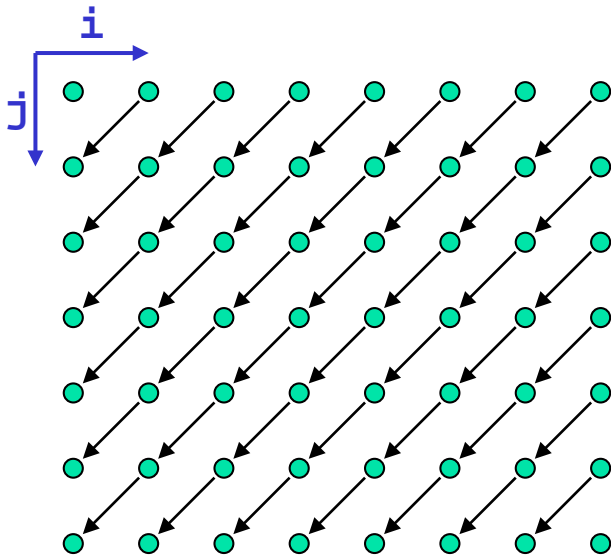
$$d_1 = (1, 0)$$
$$d_2 = (1, -1)$$



$$d'_1 = (0, 1)$$
$$d'_2 = (-1, 1)$$

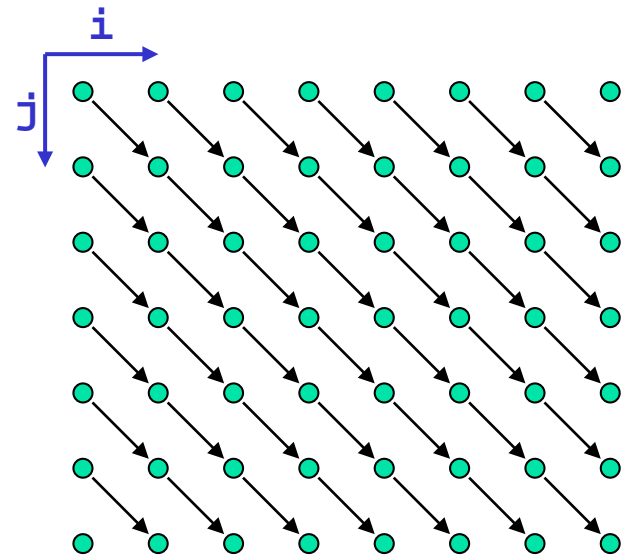
Loop Reversal

```
for (j=1; j<=N; j++)  
  for (i=1; i<=M; i++)  
    A[i,j] = F(A[i+1,j-1]);
```



$$d_1 = (1, -1)$$

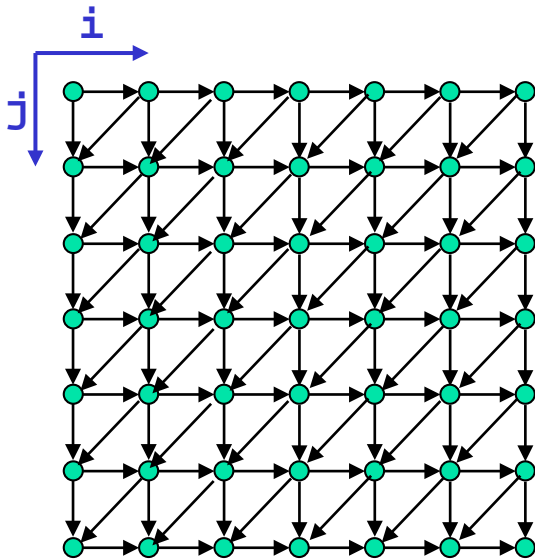
```
for (j=1; j<=N; j++)  
  for (i=-M; i<=-1; i++)  
    A[-i,j] = F(A[-i+1,j-1]);
```



$$d'_1 = (1, 1)$$

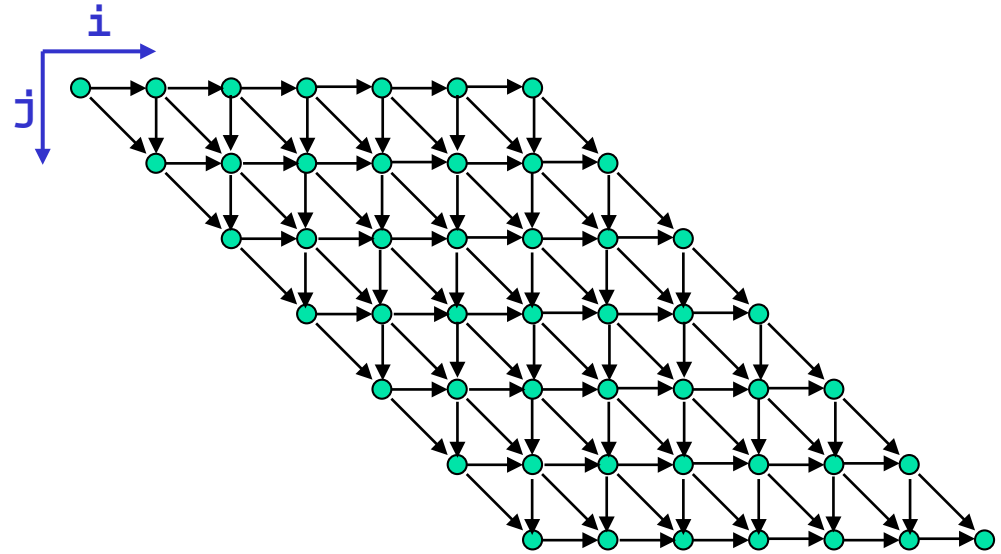
Loop Skewing

```
for (j=1; j<=N; j++)  
  for (i=1; i<=N; i++)  
    A[i,j] = F(A[i-1,j],  
              A[i,j-1],A[i+1,j-1]);
```



$$\begin{aligned}d_1 &= (0, 1) \\ d_2 &= (1, 0) \\ d_3 &= (1, -1)\end{aligned}$$

```
for (j=1; j<=N; j++)  
  for (i=j+1; i<=j+N; i++)  
    A[i-j,j] = F(A[i-j-1,j],  
                A[i-j,j-1], A[i-j+1,j-1]);
```



$$\begin{aligned}d'_1 &= (0, 1) \\ d'_2 &= (1, 1) \\ d'_3 &= (1, 0)\end{aligned}$$

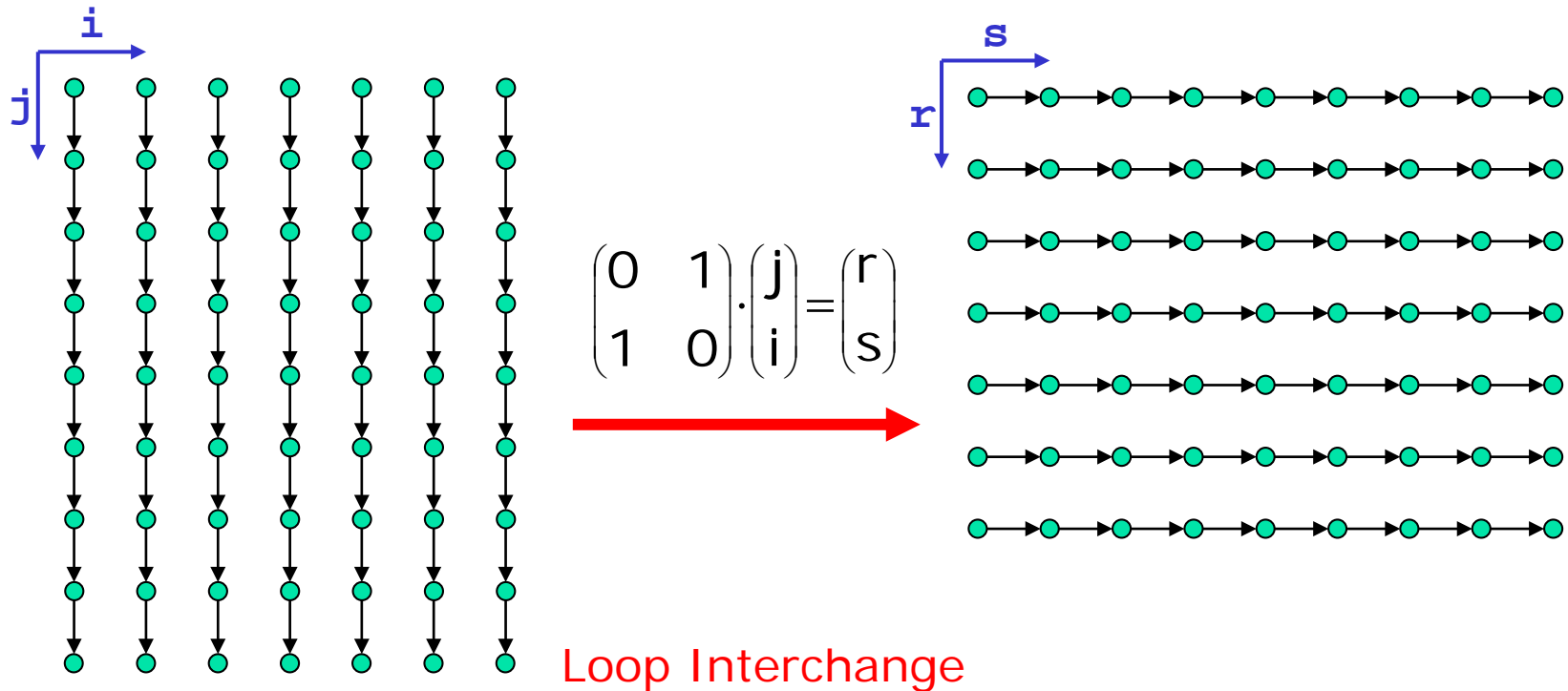
Unimodular Transformations

- Unimodular transformations can be represented by unimodular matrices
- A unimodular matrix has three properties:
 - **It is square**: it maps a n -dimensional IS to an n -dimensional IS
 - **It has all integer components**: it maps integer vectors to integer vectors
 - **Its determinant is 1 or -1**
- Combinations of unimodular loop transformations and inverses of unimodular loop transformations are also unimodular loop transformations

Unimodular Transformations

```
for (j=1; j<=M; j++)  
  for (i=1; i<=N; i++)  
    A(i,j) = F(A(i,j-1));
```

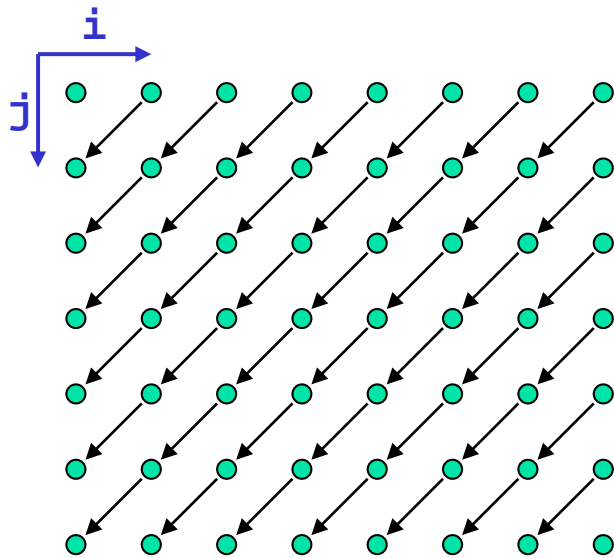
```
for (r=1; r<=N; r++)  
  for (s=1; s<=M; s++)  
    A(r,s) = F(A(r,s-1));
```



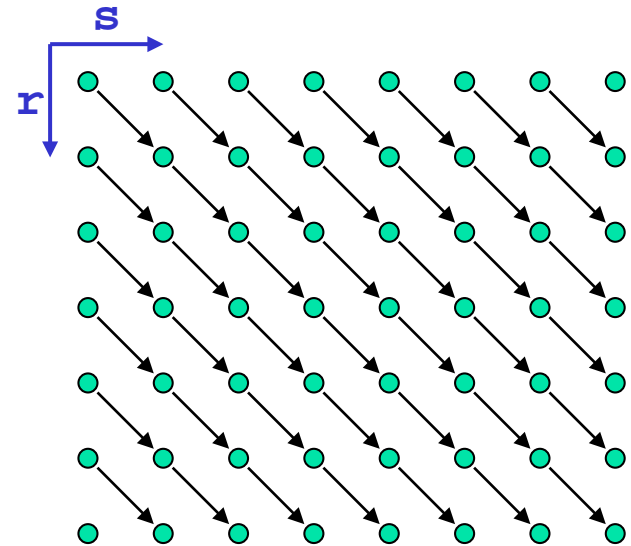
Unimodular Transformations

```
for (j=1; j<=N; j++)  
  for (i=1; i<=M; i++)  
    A[i,j] = F(A[i+1,j-1]);
```

```
for (r=1; r<=N; r++)  
  for (s=-M; s<=-1; s++)  
    A[-s,r] = F(A[-s+1,r-1]);
```



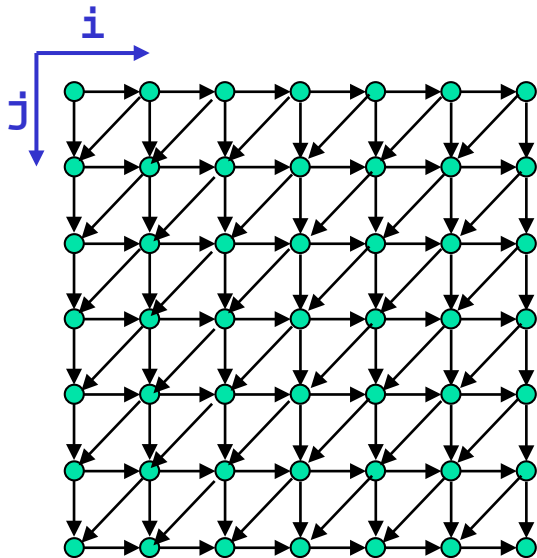
$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} r \\ s \end{pmatrix}$$



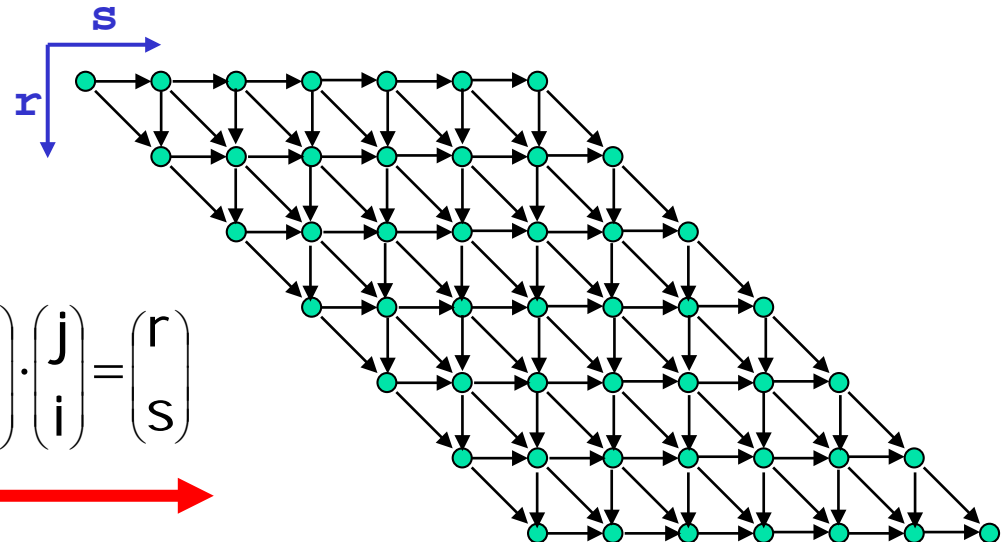
Loop Reversal

Unimodular Transformations

```
for (j=1; j<=N; j++)  
  for (i=1; i<=N; i++)  
    A[i,j] = F(A[i-1,j],  
              A[i,j-1],A[i+1,j-1]);
```



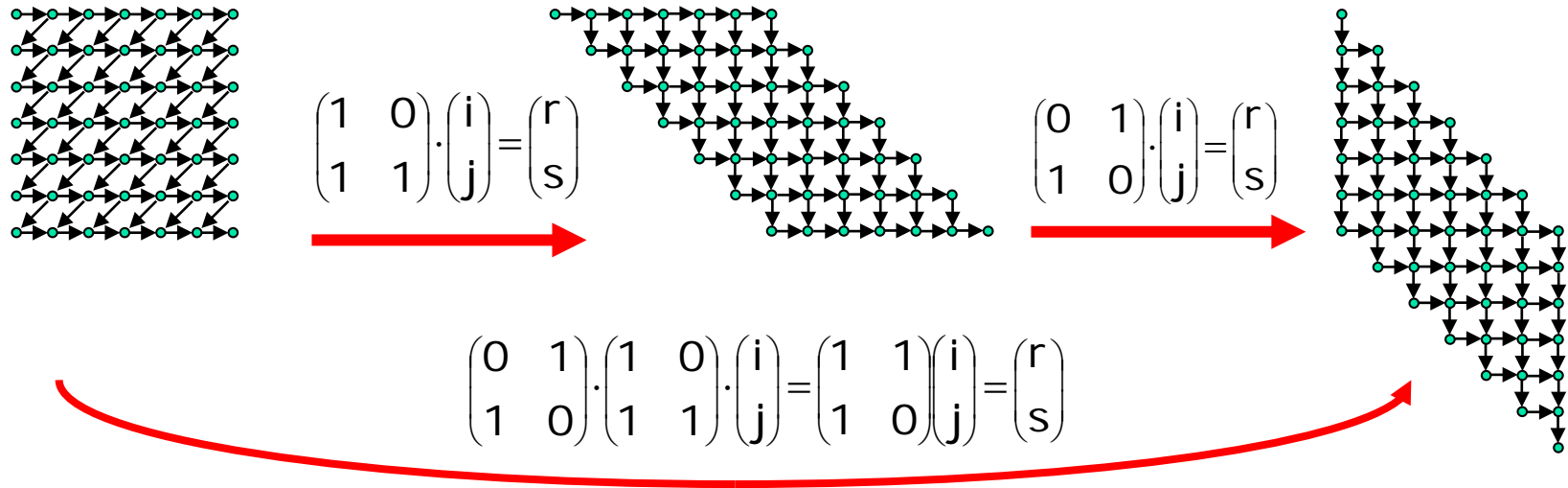
```
for (r=1; r<=N; r++)  
  for (s=r+1; s<=r+N; s++)  
    A[s-r,r] = F(A[s-r-1,r],  
                A[s-r,r-1], A[s-r+1,r-1]);
```



$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} r \\ s \end{pmatrix}$$

Loop Skewing

Unimodular Transformations



- Several transformation matrix can be combined together in a unique transformation matrix.

Unimodular Transformations

- **Legality of Unimodular Transformation**
 - Let D be the set of dependence vectors of a loop nest. A unimodular transformation T is legal if:

$$\forall d \in D : T * d \succ 0$$

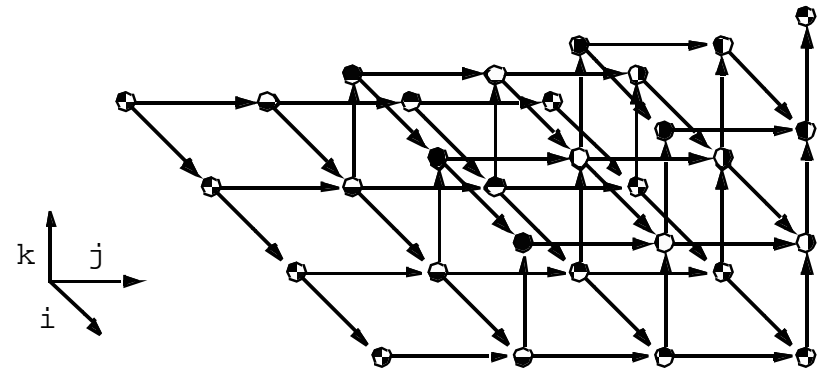
- The new dependence vectors are lexicographically positive.
- **Modifying the Program after Unimodular Transformations**
 - Rewriting the loop bounds: Fourier-Motzkin algorithm.
 - Rewriting the body of the loop nest:

$$T \cdot (i,j) = (r,s)$$

$$(i,j) = T^{-1} \cdot (r,s)$$

Example: LU Decomposition

```
for (k=0; k<=N-1; k++)  
  for (i=k; i<=N-1; i++)  
    for (j=k; j<=N-1; j++)  
      F(i,j,k)
```



$$d_1 = (1, 0, 0)$$

$$d_2 = (0, 1, 0)$$

$$d_3 = (0, 0, 1)$$

Example: LU Decomposition

```
for (k=0; k<=N-1; k++)  
  for (i=k; i<=N-1; i++)  
    for (j=k; j<=N-1; j++)  
      F(i,j,k)
```

$$d_1 = (1, 0, 0)$$

$$d_2 = (0, 1, 0)$$

$$d_3 = (0, 0, 1)$$

Unimodular Transformation

$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Legality Test

$$T \cdot d_1 = (1, 0, 1)^t$$

$$T \cdot d_2 = (0, 1, 0)^t$$

$$T \cdot d_3 = (1, 0, 0)^t$$

Dependences are
lexicographically positive!!!

The unimodular transformation is legal

Example: LU Decomposition

- Original Iteration Space

```
for (k=0; k<=N-1; k++)
  for (i=k; i<=N-1; i++)
    for (j=k; j<=N-1; j++)
      F(i,j,k)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} k \\ i \\ j \end{pmatrix} \leq \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\left\{ \begin{array}{l} A \cdot \begin{pmatrix} k \\ i \\ j \end{pmatrix} \leq \beta \\ T \cdot \begin{pmatrix} k \\ i \\ j \end{pmatrix} = \begin{pmatrix} r \\ s \\ t \end{pmatrix} \Rightarrow \begin{pmatrix} k \\ i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \end{array} \right\} \Rightarrow A \cdot T^{-1} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} = A' \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \beta$$

Example: LU Decomposition

- Transformed Iteration Space

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \quad \beta = \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

To compute the new loop bounds we apply the Fourier-Motzkin Algorithm to this set of inequalities

$$A' = A * T^{-1}$$

Example: LU Decomposition

- Fourier-Motzkin Algorithm

- 1) Separate Coefficients

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 1 \\ -1/2 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \end{pmatrix}$$

$$(0 \ 1 \ 0) \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq (N-1)$$

Example: LU Decomposition

- Fourier-Motzkin Algorithm

- 2) Compute Bounds of t

Lower Bounds of t

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} - \begin{pmatrix} N-1 \\ 0 \end{pmatrix} \leq (t)$$

$$\max(0, r - N + 1) \leq (t)$$

Upper Bounds of t

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 1 \\ -1/2 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \end{pmatrix} \quad \rightarrow \quad (t) \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ 0 & -1 \\ -1/2 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix}$$

$$(t) \leq \min(N-1, s, \lfloor r/2 \rfloor)$$

Example: LU Decomposition

- Fourier-Motzkin Algorithm
 - 3) Combine inequalities

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} - \begin{pmatrix} N-1 \\ 0 \end{pmatrix} \leq (t) \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ 0 & -1 \\ -1/2 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix}$$

$$(0 \ 1 \ 0) \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} \leq (N-1)$$

Example: LU Decomposition

- Fourier-Motzkin Algorithm

- 3) Combine inequalities

$$\rightarrow \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} - \begin{pmatrix} N-1 \\ 0 \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ 0 & -1 \\ -1/2 & 0 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix}$$

$$\rightarrow (0 \ 1) \cdot \begin{pmatrix} r \\ s \end{pmatrix} \leq (N-1)$$

Obtain a new system of inequalities:

$$\begin{pmatrix} 0 & 1 \\ 0 & -1 \\ -1 & 0 \\ 1 & 0 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} \leq \begin{pmatrix} N-1 \\ 0 \\ 0 \\ 2N-2 \\ N-1 \end{pmatrix}$$

Example: LU Decomposition

- Fourier-Motzkin Algorithm

4) Repeat for r and s

Finally, the new loop bounds are:

$$0 \leq r \leq 2N-2$$

$$\max(0, r-N+1) \leq s \leq N-1$$

$$\max(0, r-N+1) \leq t \leq \min(N-1, s, \lfloor r/2 \rfloor)$$

- Rewriting the body of the loop nest: Modifying the array index functions by performing this substitution:

$$\begin{pmatrix} k \\ i \\ j \end{pmatrix} \quad \text{by} \quad T^{-1} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} t \\ s \\ r-t \end{pmatrix}$$

Example: LU Decomposition

Source Code

```
for (k=0; k<=N-1; k++)  
    for (i=k; i<=N-1; i++)  
        for (j=k; j<=N-1; j++)  
            F(i,j,k)
```

$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Transformed Code

```
for (r=0; r<=2*N-2; r++)  
    for (s=max(0,r-N+1); s<=N-1; s++)  
        for (t=max(r-N+1,0); r<=min(N-1,s,Down(r/2)); r++)  
            F(s,r-t,t)
```

Exercise

- Give each of the unimodular matrix transformations T that we need to apply to the following loop nest to achieve each of the six different forms of the triangular matrix multiplication code.

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

Exercise

- Give each of the unimodular matrix transformations T that we need to apply to the following loop nest to achieve each of the six different forms of the triangular matrix multiplication code.

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

$$T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



KJI form

$$T_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



KIJ form

$$T_3 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



JKI form

$$T_4 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$



JIK form

$$T_5 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$



IKJ form

$$T_6 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$



IJK form

Exercise

- Compute the loop bounds of the IJK form using Fourier-Motzkin algorithm.

```
float A[N][N], B[N][N], C[N][N];  
for (i=..... )  
    for (j=..... )  
        for (k=..... )  
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Exercise

- Compute the loop bounds of the IJK form using Fourier-Motzkin algorithm.

```
float A[N][N], B[N][N], C[N][N];  
for (i=0; i<=N-1; i++)  
    for (j=0; j<=N-1; j++)  
        for (k=0; k<=min(N-1,j,i); k++)  
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

Loop Fusion

- Loop fusion takes **two adjacent loops with same IS** and combines their loop bodies **into a single loop**
- Loop fusion is legal if:
 - The loops have the same bounds
 - There are no dependences in the fused loop for which statements from the first loop depend on statements from the second loop (that is, no data dependences are reversed)
- Loop fusion may improve reuse by:
 - moving accesses to the same cache line to the same loop iteration
 - providing a perfect nest that enables a loop permutation with better locality
- Using reuse analysis, compilers can determine the profitability of fusing two loop nest

Loop Fusion

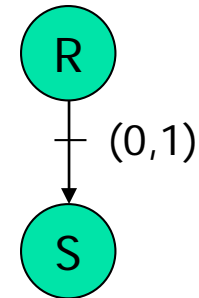
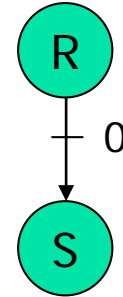
```
for (j=1; j<=M; j++)  
{  
  for (i=1; i<=N; i++)  
    R: A[i,j] = B[i,j]+C[i,j];  
  for (i=1; i<=N; i++)  
    S: B[i-1,j] = C[i,j]+1.5;  
}
```

Loop Fusion

```
for (j=1; j<=M; j++)  
{  
  for (i=1; i<=N; i++)  
    R: A[i,j] = B[i,j]+C[i,j];  
    S: B[i-1,j] = C[i,j]+1.5;  
}
```

Loop Interchange

```
for (i=1; i<=N; i++)  
{  
  for (j=1; j<=M; j++)  
    R: A[i,j] = B[i,j]+C[i,j];  
    S: B[i-1,j] = C[i,j]+1.5;  
}
```

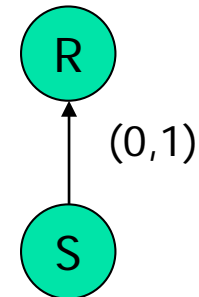
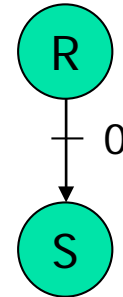


Loop Fusion

```
for (j=0; j<=M; j++)  
{  
  for (i=0; i<=N; i++)  
    R: A[i,j] = B[i,j]+C[i,j];  
  for (i=0; i<=N; i++)  
    S: B[i+1,j] = C[i,j]+1.5;  
}
```

Loop Fusion

```
for (j=0; j<=M; j++)  
{  
  for (i=0; i<=N; i++)  
    R: A[i,j] = B[i,j]+C[i,j];  
    S: B[i+1,j] = C[i,j]+1.5;  
}
```



- Loop fusion is not legal!!!

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

Loop Distribution

- Loop Distribution takes a loop that contains multiple statements and splits it into two loops with the same IS
- Loop Distribution is legal if it does not result in breaking any cycles in the dependence graph of the original loop
- Loop Distribution is effective for improving reuse by enabling loop permutation on a nest that is not permutable

Loop Distribution

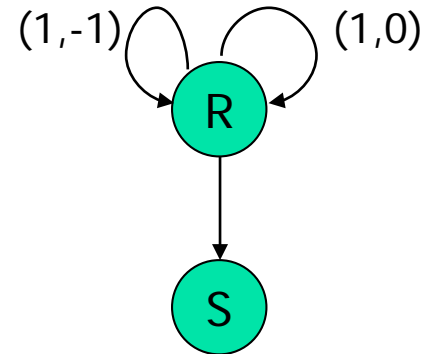
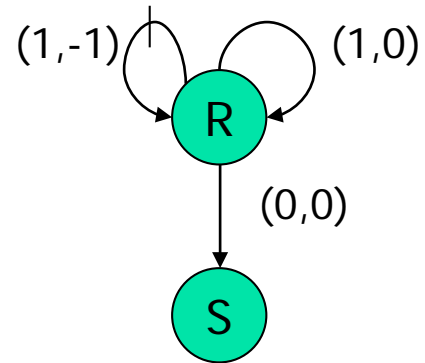
```
for (j=1; j<=M; j++)  
  for (i=1; i<=N; i++)  
  {R: A[i,j] = A[i,j-1]+A[i-1,j+1];  
   S: B[i,j] = A[i,j]+C[i,j];  
  }
```

Loop Distribution

```
for (j=1; j<=M; j++)  
  for (i=1; i<=N; i++)  
    R: A[i,j] = A[i,j-1]+A[i-1,j+1];  
  for (j=1; j<=M; j++)  
    for (i=1; i<=N; i++)  
      S: B[i,j] = A[i,j]+C[i,j];
```

Loop Interchange in 2nd loop

```
. . .  
for (i=1; i<=N; i++)  
  for (j=1; j<=M; j++)  
    S: B[i,j] = A[i,j]+C[i,j];
```



Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- **Scalar Replacement**
- Unroll & Jam
- Tiling
- Combination of transformations

Scalar Replacement

- Most compilers fail to recognize opportunities for reuse of array variables
- Example

```
for (i=0; i<=N-1; i++)  
    for (j=0; j<=N-1; j++)  
        A[i]= A[i]+B[j];
```

```
for (i=0; i<=N-1; i++){  
    R= A[i];  
    for (j=0; j<=N-1; j++)  
        R= R+B[j];  
    A[i]= R;  
}
```

- **Scalar Replacement** is a transformation that uses dependence information to find reuse of array values and expose it by replacing the references with scalar temporaries

Scalar Replacement

- The original and the transformed codes can be more complex

```
for (i=0; i<=N-1; i++){  
    A[i+2] = A[i] + C[i];  
    B[k] = B[k] + C[i];  
}
```

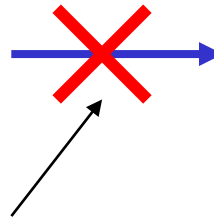
We save 2/3 of the memory accesses. For $N=900$, the transformed code performs 1800 accesses instead of the 5400 accesses of the original code.

```
t1 = b[k];  
q0 = a[0];  
q1 = a[1];  
for (i=0; i<=N-1; i=i+2){  
    t2 = c[i];  
    q0 = q0 + t2;  
    A[i+2] = q0;  
    t1 = t1 + t2;  
    t2 = c[i+1];  
    q1 = q1 + t2;  
    A[i+3] = q1;  
    t1 = t1 + t2;  
}  
B[k] = t1;
```

Scalar Replacement

- Scalar replacement is **not** always possible

```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=N-1; j++)  
    A[i] = A[i] + A[j];
```



```
for (i=0; i<=N-1; i++){  
  R = A[i];  
  for (j=0; j<=N-1; j++)  
    R = R + A[j];  
  A(i) = R;  
}
```

Error when $i=j$

- Loop interchange may make scalar replacement applicable to or more effective for a given loop nest:

```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=N-1; j++)  
    A[j] = A[j] + B[i, j];
```



```
for (j=0; j<=N-1; j++)  
  for (i=0; i<=N-1; i++)  
    A[j] = A[j] + B[i, j];
```

Be careful with
spatial locality!!!

Scalar Replacement

- The algorithm has 5 step:
 - 1) Determine number of temporary variables
 - 2) Reference Replacement
 - 3) Register Copying
 - 4) Code Motion
 - 5) Initialization
- 5') Register Subsumption
- The details of the transformation can be found in:
 - S. Carr and K. Kennedy. *Compiling Scientific Code for Complex Memory Hierarchies*. Twenty-Fourth Annual Hawaii International Conference on System Sciences, 1991.
 - S. Carr, D. Callahan and K. Kennedy. *Improving Register Allocation for Subscripted Variables*. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI 90), 1990.

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

Unroll and Jam

- The profitability of Scalar Replacement can be limited
- Unroll and Jam increases the opportunities for scalar replacement

```
for (i=0; i<=N-1; i++)  
    for (j=0; j<=N-1; j++)  
        v[i]= v[i]+A[i,j]*w[j];
```



```
for (i=0; i<=N-1; i++){  
    R = v[i];  
    for (j=0; j<=N-1; j++)  
        R= R+A[i,j]*w[j];  
    v[i] = R;  
}
```

In this case the
profitability of Scalar
Replacement is poor

Unroll and Jam

```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=N-1; j++)  
    v[i]= v[i]+A[i,j]*w[j];
```



Unroll the outermost loop

```
for (i=0; i<=N-1; i=i+4) {  
  for (j=0; j<=N-1; j++)  
    v[i  ]= v[i  ]+A[i  ,j]*w[j];  
  for (j=0; j<=N-1; j++)  
    v[i+1]= v[i+1]+A[i+1,j]*w[j];  
  for (j=0; j<=N-1; j++)  
    v[i+2]= v[i+2]+A[i+2,j]*w[j];  
  for (j=0; j<=N-1; j++)  
    v[i+3]= v[i+3]+A[i+3,j]*w[j];  
}
```

For simplicity, we assume N is multiple of 4. If not, the remaining iterations ($N\%4$) must be done separately.

Unrolling is always legal.

Unroll and Jam

```
for (i=0; i<=N-1; i=i+4) {  
    for (j=0; j<=N-1; j++)  
        v[i]= v[i]+A[i,j]*w[j];  
    for (j=0; j<=N-1; j++)  
        v[i+1]= v[i+1]+A[i+1,j]*w[j];  
    for (j=0; j<=N-1; j++)  
        v[i+2]= v[i+2]+A[i+2,j]*w[j];  
    for (j=0; j<=N-1; j++)  
        v[i+3]= v[i+3]+A[i+3,j]*w[j];  
}
```

Jam

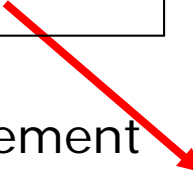
In this code there are lots of opportunities for scalar replacement.

```
for (i=0; i<=N-1; i=i+4)  
    for (j=0; j<=N-1; j++) {  
        v[i  ]= v[i  ]+A[i  ,j]*w[j];  
        v[i+1]= v[i+1]+A[i+1,j]*w[j];  
        v[i+2]= v[i+2]+A[i+2,j]*w[j];  
        v[i+3]= v[i+3]+A[i+3,j]*w[j];  
    }
```

Unroll and Jam

```
for (i=0; i<=N-1; i=i+4)
  for (j=0; j<=N-1; j++) {
    v[i] = v[i] + A[i][j]*w[j];
    v[i+1] = v[i+1] + A[i+1][j]*w[j];
    v[i+2] = v[i+2] + A[i+2][j]*w[j];
    v[i+3] = v[i+3] + A[i+3][j]*w[j];
  }
```

Scalar
Replacement



```
for (i=0; i<=N-1; i=i+4){
  R0 = v[i];
  R1 = v[i+1];
  R2 = v[i+2];
  R3 = v[i+3];
  for (j=0; j<=N-1; j++) {
    T = w[j];
    R0 = R0 + A[i][j]*T;
    R1 = R1 + A[i+1][j]*T;
    R2 = R2 + A[i+2][j]*T;
    R3 = R3 + A[i+3][j]*T;
  }
  v[i] = R0;
  v[i+1] = R1;
  v[i+2] = R2;
  v[i+3] = R3;
}
```

	Loads (*)	Stores (*)
Original code	$3N^2$	N^2
SR	$2N^2 + N$	N
Unroll&Jam + SR	$N^2 + N^2/4 + N$	N

(*) We do not consider here the level in the memory hierarchy from where they are served.

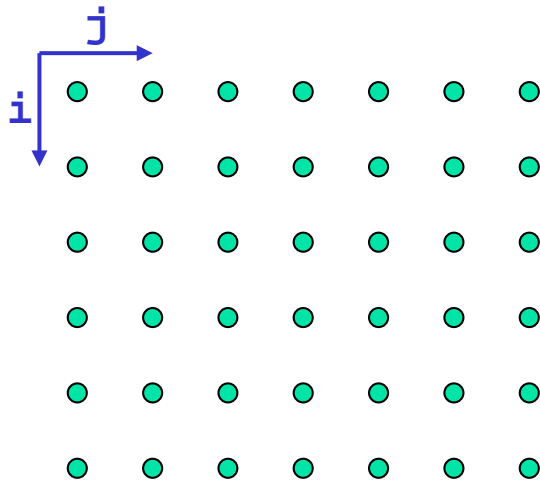
Unroll and Jam

- Legality of Unroll and Jam

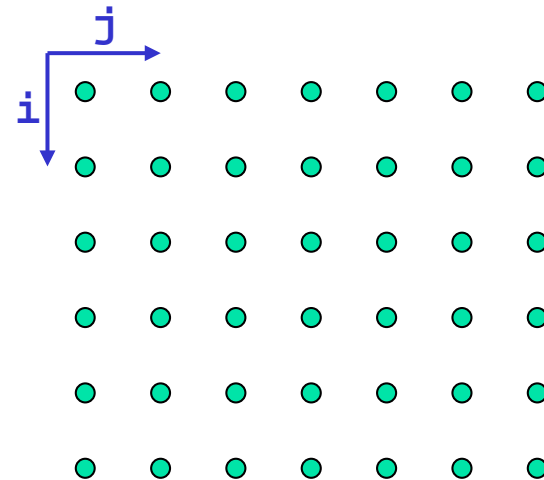
```
for (i=0; i<=6; i++)  
  for (j=0; j<=6; j++)  
    F(A[i,j]);
```

Unroll

```
for (i=0; i<=6; i=i+2){  
  for (j=0; j<=6; j++)  
    F(A[i,j]);  
  for (j=0; j<=6; j++)  
    F(A[i+1,j]);  
}
```



IS traversal is
not modified!!



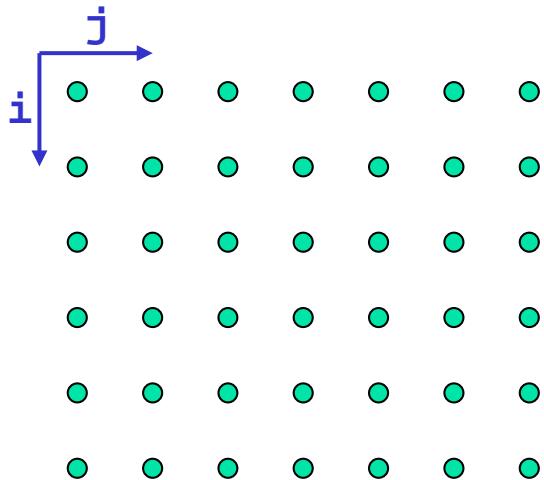
Unroll and Jam

- Legality of Unroll and Jam

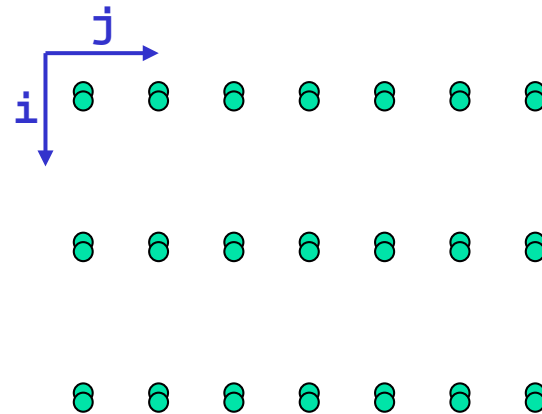
```
for (i=0; i<=6; i=i+2){  
  for (j=0; j<=6; j++)  
    F(A[i,j]);  
  for (j=0; j<=6; j++)  
    F(A[i+1,j]);  
}
```

Jam

```
for (i=0; i<=6; i=i+2)  
  for (j=0; j<=6; j++)  
  {  
    F(A[i ,j]);  
    F(A[i+1,j]);  
  }
```

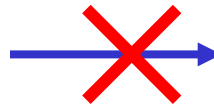
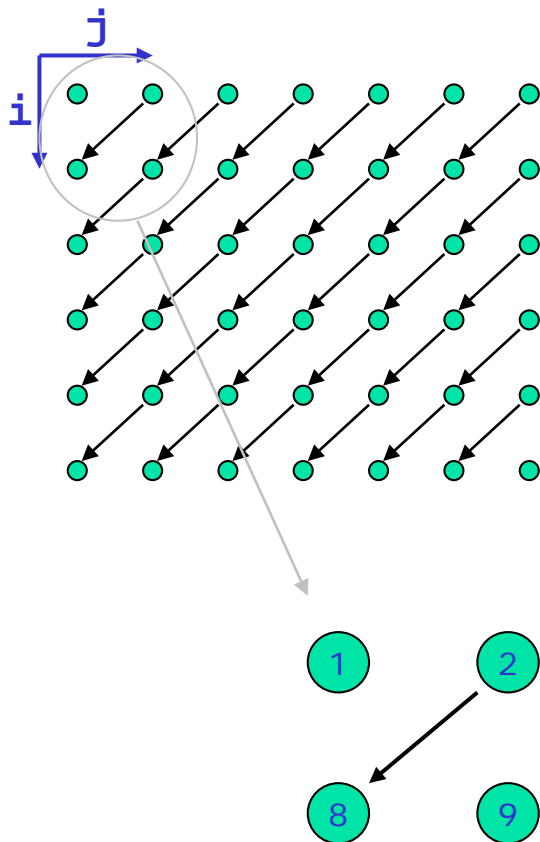


IS traversal is
modified

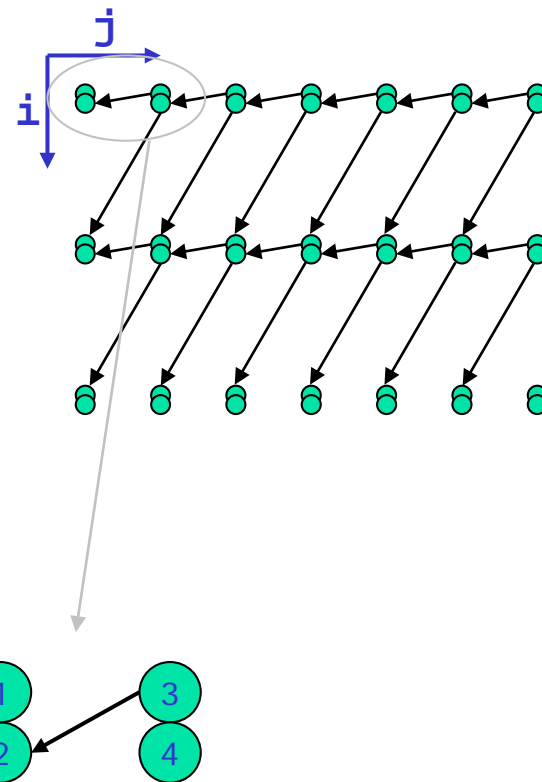


Unroll and Jam

- Legality of Unroll and Jam
 - In this case, Unroll and Jam is not legal

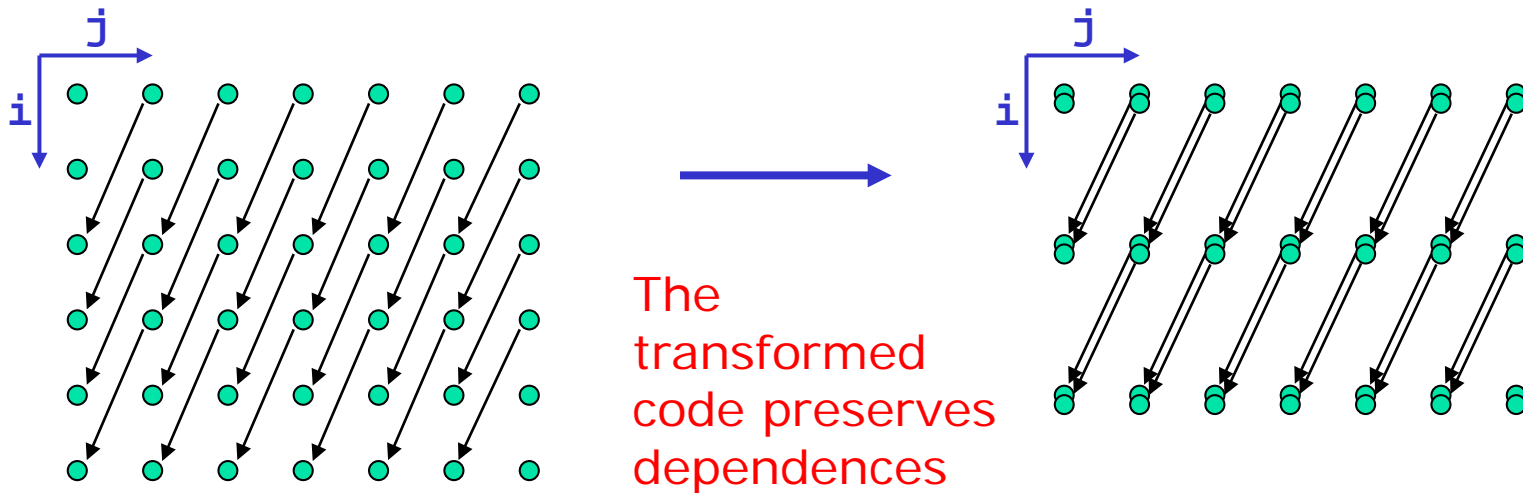


The transformed code does not preserve dependences



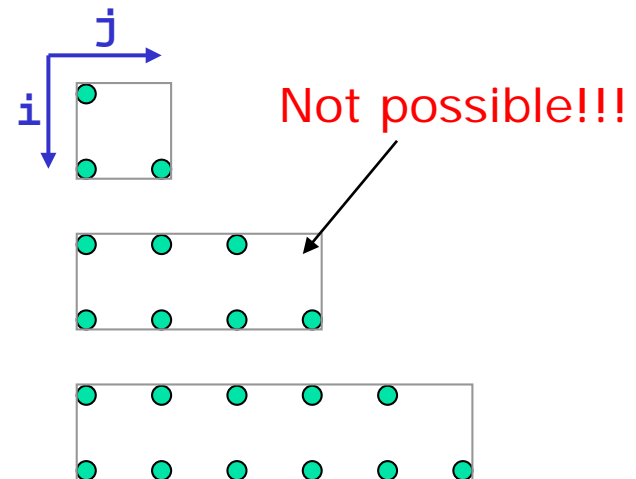
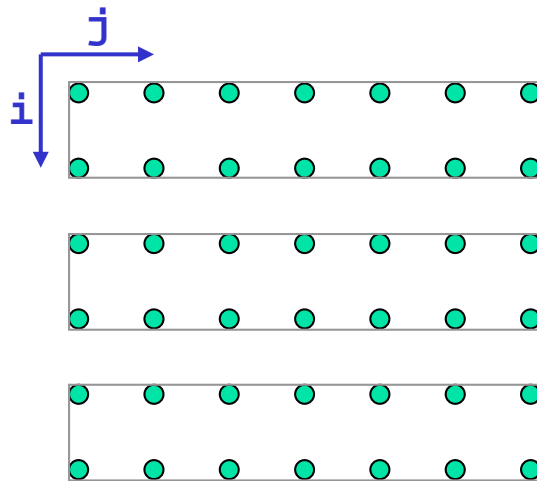
Unroll and Jam

- Legality of Unroll and Jam
 - In this case, unroll and jam is legal (max unroll factor = 2)



Unroll and Jam

- How do you select the unroll factor?
 - Legality
 - Control register pressure
 - Loop Balance
- ¿What happens in non-rectangular IS?



Unroll and Jam

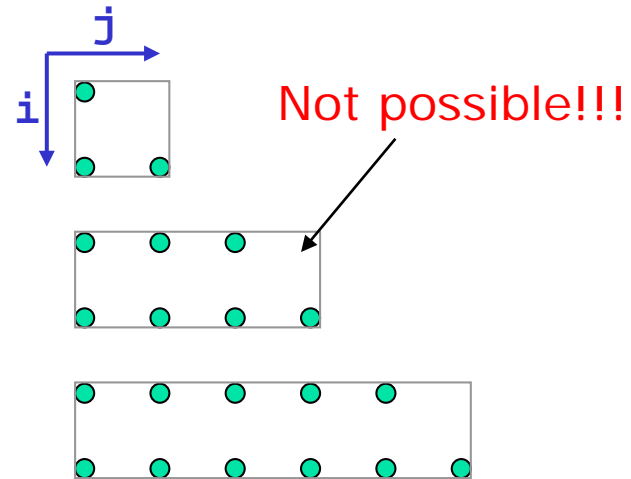
- ¿What happens in non-rectangular IS?

```
for (i=0; i<=5; i++)  
  for (j=0; j<=i; j++)  
    F(A([i,j]));
```

Unroll

```
for (i=0; i<=5; i=i+2){  
  for (j=0; j<=i; j++)  
    F(A([i,j]));  
  for (j=0; j<=i+1; j++)  
    F(A[i+1,j]);  
}
```

Jamming is not possible!!
The bounds of j-loops are
different!!!



To perform unroll and jam
it is necessary to perform
another transformation
previously

Unroll and Jam

- ¿What happens in non-rectangular IS?

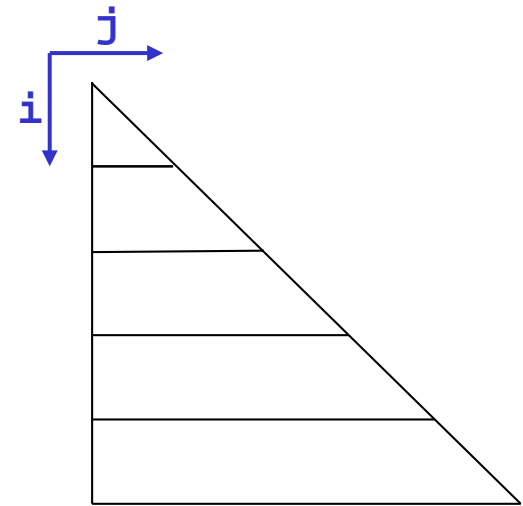
```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=i; j++)  
    F(A[i,j]);
```

Strip mining

```
for (ii=0; ii<=N-1; ii=ii+2)  
  for (i=ii; i<=ii+1; i++)  
    for (j=0; j<=i; j++)  
      F(A[i,j]);
```

Loop Interchange

```
for (ii=0; ii<=N-1; ii=ii+2)  
  for (j=0; j<=ii+1; j++)  
    for (i=max(ii,j); i<=ii+1; i++)  
      F(A[i,j]);
```

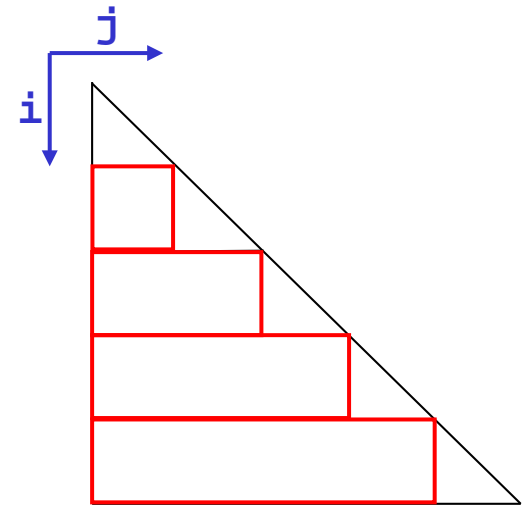


Unroll and Jam

- ¿What happens in non-rectangular IS?

```
for (ii=0; ii<=N-1; ii=ii+2)
  for (j=0; j<=ii+1; j++)
    for (i=max(ii,j); i<=ii+1; i++)
      F(A[i,j]);
```

Index Set
Splitting



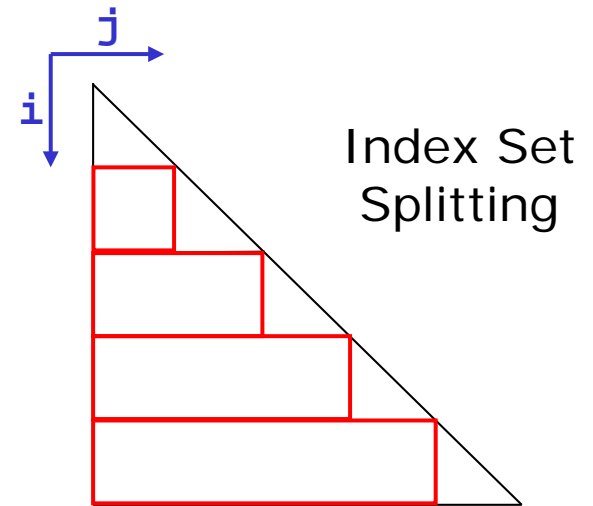
```
for (ii=0; ii<=N-1; ii=ii+2){
  for (j=0; j<=ii; j++)
    for (i=ii; i<=ii+1; i++)
      F(A[i,j]);
  for (j=ii+1; j<=ii+1; j++)
    for (i=j; i<=ii+1; i++)
      F(A[i,j]);
}
```

Unroll and Jam

- ¿What happens in non-rectangular IS?

```
for (ii=0; ii<=N-1; ii=ii+2)
  for (j=0; j<=ii+1; j++)
    for (i=max(ii,j); i<=ii+1; i++)
      F(A[i,j]);
```

```
for (ii=0; ii<=N-1; ii=ii+2){
  for (j=0; j<=ii; j++)
    for (i=ii; i<=ii+1; i++)
      F(A[i,j]);
  for (j=ii+1; j<=ii+1; j++)
    for (i=j; i<=ii+1; i++)
      F(A[i,j]);
}
```



This loop can be unrolled twice. It is equivalent to Unroll and Jam

Unroll and Jam

- Putting all together

```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=i; j++)  
    F(A[i,j]);
```

Strip mining

```
for (ii=0; ii<=N-1; ii=ii+2)  
  for (i=ii; i<=ii+1; i++)  
    for (j=0; j<=i; j++)  
      F(A[i,j]);
```

Loop Interchange

```
for (ii=0; ii<=N-1; ii=ii+2)  
  for (j=0; j<=ii+1; j++)  
    for (i=max(ii,j); i<=ii+1; i++)  
      F(A[i,j]);
```

Index Set
Splitting

```
for (ii=0; ii<=N-1; ii=ii+2){  
  for (j=0; j<=ii; j++)  
    for (i=ii; i<=ii+1; i++)  
      F(A[i,j]);  
  for (j=ii+1; j<=ii+1; j++)  
    for (i=j; i<=ii+1; i++)  
      F(A[i,j]);  
}
```

Unroll

```
for (ii=0; ii<=N-1; ii=ii+2){  
  for (j=0; j<=ii; j++){  
    F(A[ii,j]);  
    F(A[ii+1,j]);  
  }  
  for (j=ii+1; j<=ii+1; j++)  
    for (i=j; i<=ii+1; i++)  
      F(A[i,j]);  
}
```

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

Tiling

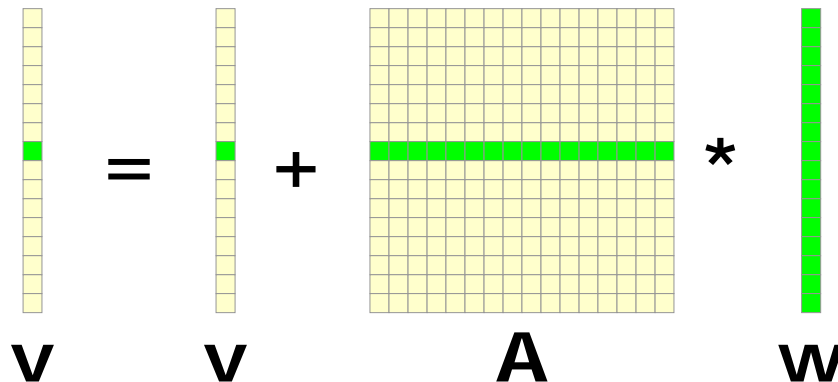
- Tiling is a loop transformation that a compiler can use to automatically create block algorithms.
 - Enhance data locality
 - Enhance parallelism
- Tiling consists of dividing the original IS into regular blocks/tiles, creating a blocking of the data arrays
- The idea is to shorten the distance between successive references to the same memory location, so that the probability of finding the associated data in the memory level being exploited is higher
- Multi-dimensional tiling and Multilevel tiling

Tiling

- Example: matrix-vector multiplication

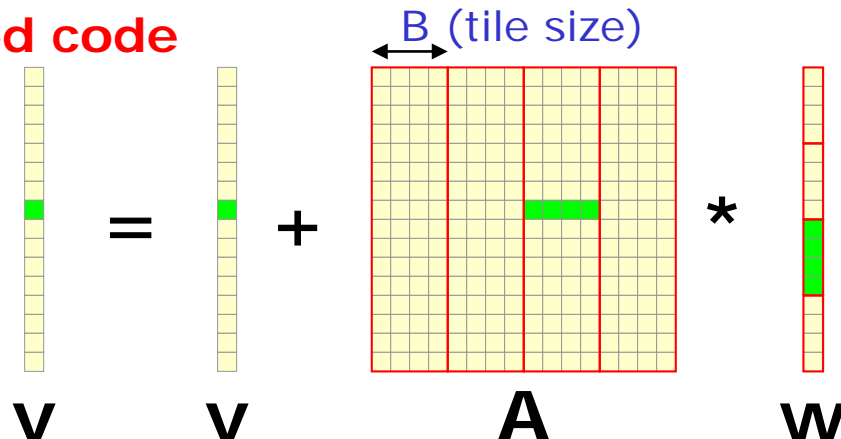
```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=M-1; j++)  
    v[i]=v[i]+A[i][j]*w[j];
```

Original code



- There is spatial locality in array references $A[i][j]$ and $w[j]$.
- Temporal locality can be exploited by allocating $v[i]$ in a register (scalar replacement).
- If w does not fit in cache, temporal reuse of w cannot be exploited.

Tiled code



- If cache can hold B elements of w , the tiled code exploit temporal reuse of w .

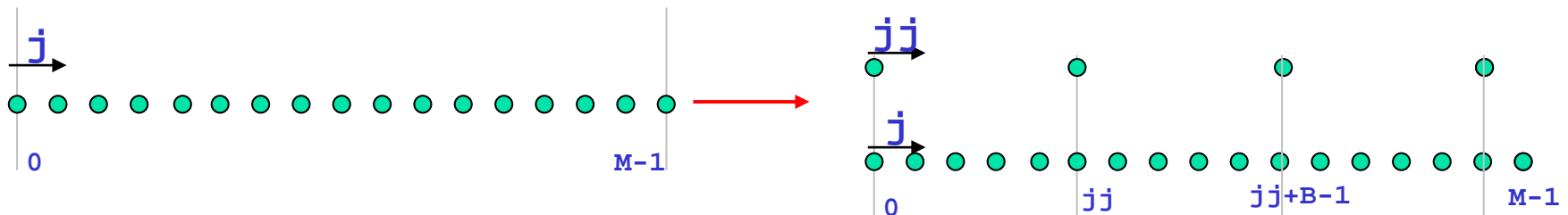
Tiling

- Loop Tiling can be implemented as a combination of Strip-mining and loop permutation transformation:
 - Strip-mining: applies to the outermost loop, a previous loop permutation is sometimes necessary

```
for (j=0; j<=M-1; j++)  
  . . .
```

Strip-mining is
always legal

```
for (jj=0; jj<=M-1; jj+=B)  
  for (j=jj; j<=min(jj+B-1, M-1); j++)  
    . . .
```



Tiling

- Loop permutation:
 - Used to establish the order in which iterations inside the tiles are traversed.
 - The loops involved in the permutation are always the innermost loops that have steps equal to 1. Therefore, the theory of unimodular transformation can be used to perform the loop permutation.

```
• • •  
  for (j=jj; j<=min(jj+B-1,M-1); j++)  
    for (i=0; i<=N-1; i++)  
      • • •
```

```
• • •  
  for (i=0; i<=N-1; i++)  
    for (j=jj; j<=min(jj+B-1,M-1); j++)  
      • • •
```

Loop
Permutation



Tiling

```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=M-1; j++)  
    v[i]=v[i]+A[i][j]*w[j];
```

Loop
permutation

```
for (j=0; j<=M-1; j++)  
  for (i=0; i<=N-1; i++)  
    v[i]=v[i]+A[i][j]*w[j];
```

Strip-mining

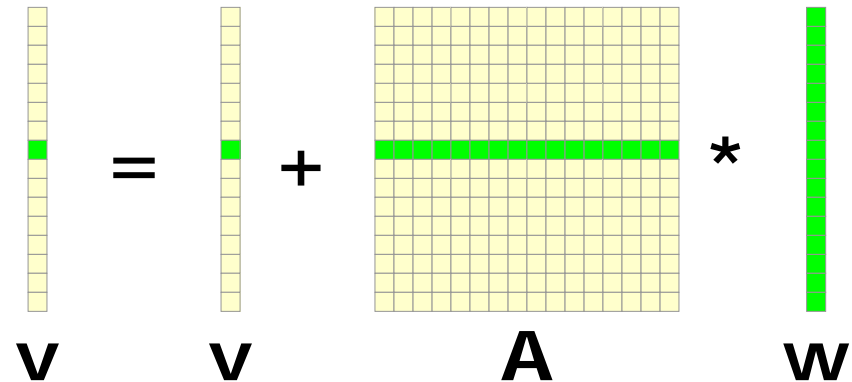
```
for (jj=0; jj<=M-1; jj+=B)  
  for (j=jj; j<=min(jj+B-1,M-1); j++)  
    for (i=0; i<=N-1; i++)  
      v[i]=v[i]+A[i][j]*w[j];
```

Loop
Permutation

```
for (jj=0; jj<=M-1; jj+=B)  
  for (i=0; i<=N-1; i++)  
    for (j=jj; j<=min(jj+B-1,M-1); j++)  
      v[i]=v[i]+A[i][j]*w[j];
```

Tiling

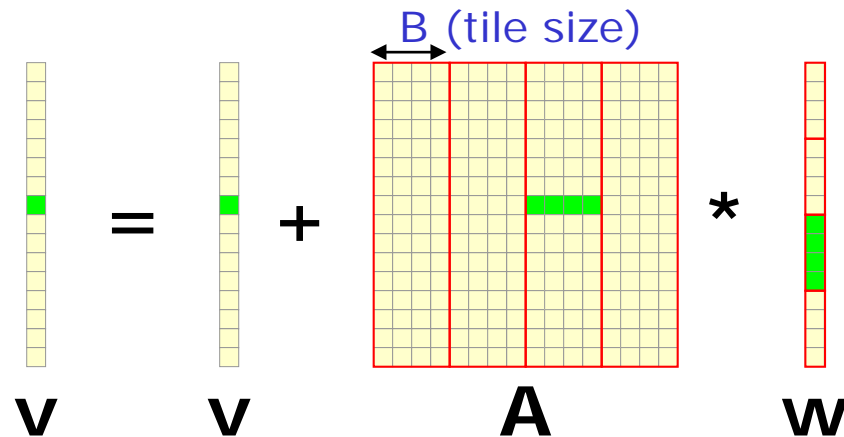
```
for (i=0; i<=N-1; i++)  
  for (j=0; j<=M-1; j++)  
    v[i]=v[i]+A[i][j]*w[j];
```



- Assume: w does not fit in cache, cache is fully associative and each cache line can hold cls elements.
 - Accesses to v produce N/cls cache misses
 - Accesses to A produce $N \cdot M/cls$ cache misses
 - Accesses to w produce $N \cdot M/cls$ cache misses

Tiling

```
for (jj=0; jj<=M-1; jj+=B)
  for (i=0; i<=N-1; i++)
    for (j=jj; j<=min(jj+B-1,M-1); j++)
      v[i]=v[i]+A[i][j]*w[j];
```



- Assume cache can hold B (tile size) elements of w :
 - Accesses to v produce $(M/B) \cdot N/\text{cls}$ cache misses
 - Accesses to A produce $N \cdot M/\text{cls}$ cache misses
 - Accesses to w produce M/cls cache misses

Tiling

- Implementation for the register level:
 - Needs an extra phase (not needed for other memory levels)
 - fully unrolling + scalar replacement
- Implementation of Multi-dimensional Tiling :
 - IS has to be partitioned in more than one dimension for a particular memory level
 - Apply strip-mining and loop permutation repeatedly as many times as dimensions have to be partitioned
- Implementation of Multilevel Tiling:
 - Several levels of the memory hierarchy are exploited
 - Apply tiling level by level, dividing a tile of higher level into subtiles

Tiling

- Determining Tiling parameters:
 - The loops to be tiled at each level
 - The relative order of the loops.
 - Tile sizes.
- Information to take into account:
 - Memory level being exploited (Cache or registers)
 - Organization of the memory level (associativity, direct-mapped,...)
 - Memory size (cache size, #registers, TLB size...)

Tiling

- Example:
 - Original loop: I, J, K
 - Tiled loop: III JJJ KK II J I K
 - Multilevel tiling: 2 levels
 - Multidimensional tiling: 2 dimensions in each level
 - Tile sizes: B_{III} , B_{JJJ} , B_{KK} , B_{II}
 - Strip-mining & loop permutation is performed in this order:
I J K \rightarrow III I J K \rightarrow III J I K \rightarrow III JJJ J I K \rightarrow III JJJ K I J \rightarrow III
JJJ KK K I J \rightarrow III JJJ KK I K J \rightarrow III JJJ KK II I K J \rightarrow III JJJ
KK II J I K

Tiling

- Legality of Tiling
 - Tiling is legal iff the original loop nest is fully permutable
 - A loop nest is fully permutable if all possible permutations of the loops are legal. This means that all components of all dependence vectors are positive or 0.
 - Example: Let $D = \{(1,0), (0,1), (1,1)\}$, then the loop nest is fully permutable
 - Example: Let $D = \{(1,0), (0,1), (1,-1)\}$, then the loop nest is not fully permutable
 - However, loop skewing can be applied and dependences become $D' = \{(1,1), (0,1), (1,0)\} \rightarrow$ fully permutable
 - Non-fully permutable loop nest can always be turned into fully permutable nests by applying loop skewing

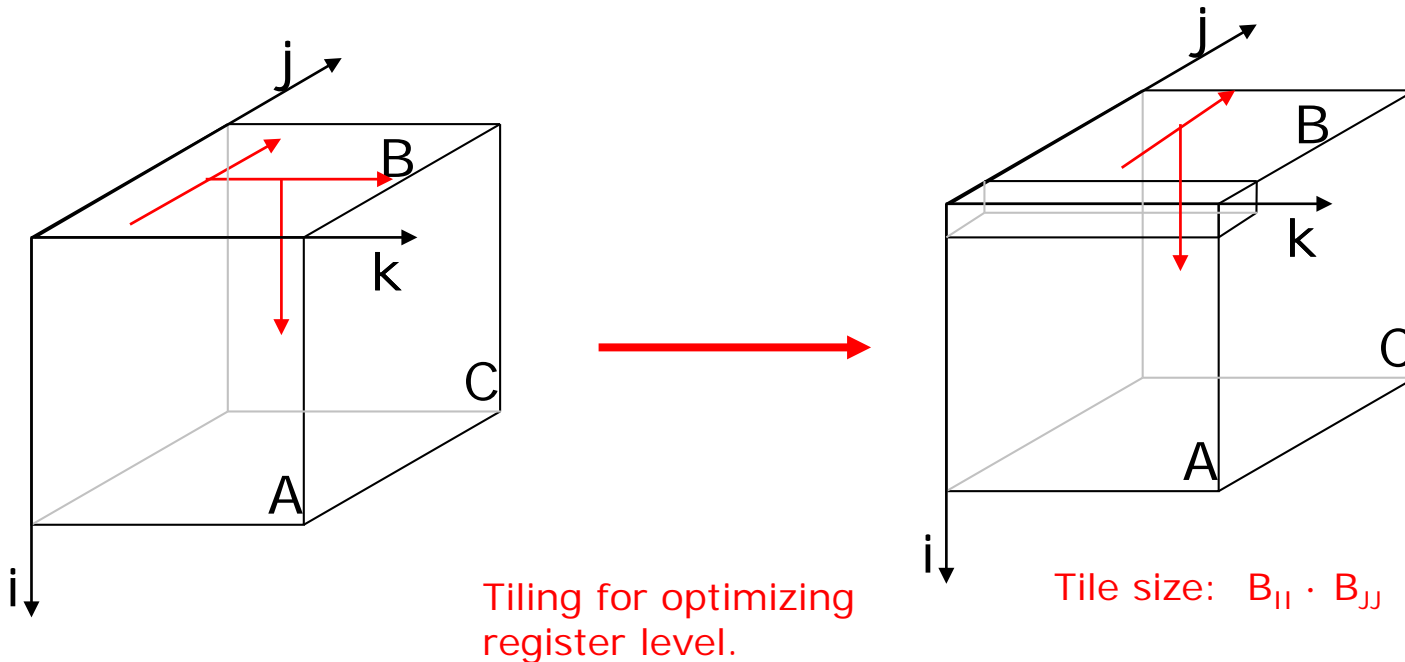
Tiling

- Effects of Tiling
 - Tiling only for the register level
 - Improves ILP
 - Reduces load/store instructions
 - Can increase TLB misses
 - Tiling only for the cache level
 - Reduces capacity cache misses
 - Tiling for cache and register levels
 - Reduces capacity cache misses
 - Improves ILP
 - Reduces load/store instructions
 - Moderate TLB misses

Tiling

- Example: Matrix-matrix multiplication, tiling for the register level

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



Tiling

- Example: Matrix-matrix multiplication, tiling for the register level

```
for (II=0; II<N; II+=2)
  for (JJ=0; JJ<N; JJ+=2)
    for (k=0; k<N; k++)
      for (i=II; i<=II+1; i++)
        for (j=JJ; j<=JJ+1; j++)
          C[i][j] = C[i][j]+A[i][k]*B[k][j];
```

```
for (II=0; II<N; II+=2)
  for (JJ=0; JJ<N; JJ+=2)
    for (k=0, i=II, j=JJ; k<N; k++) {
      C[i][j] = C[i][j]+A[i][k]*B[k][j];
      C[i+1][j] = C[i+1][j]+A[i+1][k]*B[k][j];
      C[i][j+1] = C[i][j+1]+A[i][k]*B[k][j+1];
      C[i+1][j+1] = C[i+1][j+1]+A[i+1][k]*B[k][j+1];
    }
```

Tiling

- Example: Matrix-matrix multiplication, tiling for the register level

```
for (II=0; II<N; II+=2)
  for (JJ=0; JJ<N; JJ+=2)
    for (k=0, i=II, j=JJ; k<N; k++) {
      C[i][j] = C[i][j]+A[i][k]*B[k][j];
      C[i+1][j] = C[i+1][j]+A[i+1][k]*B[k][j];
      C[i][j+1] = C[i][j+1]+A[i][k]*B[k][j+1];
      C[i+1][j+1] = C[i+1][j+1]+A[i+1][k]*B[k][j+1];
    }
```

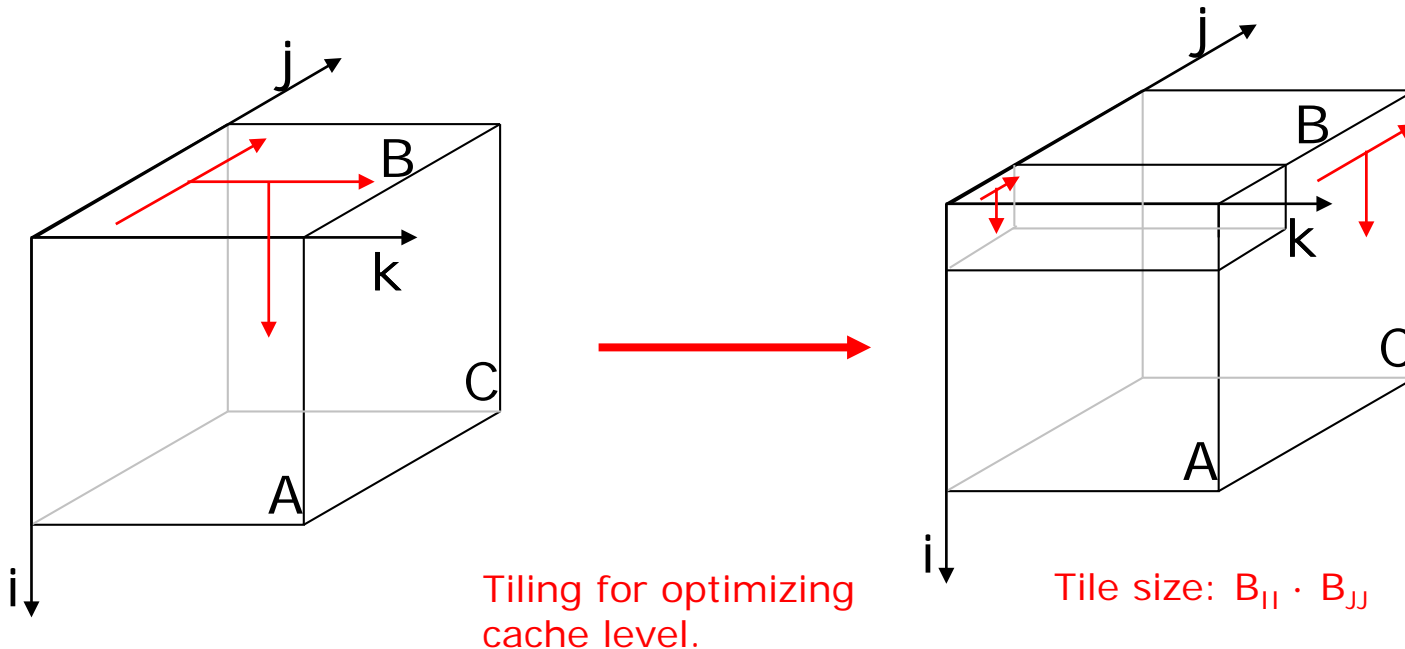
```
for (II=0; II<N; II+=2)
  for (JJ=0; JJ<N; JJ+=2){
    T00=C[II][JJ];T01=C[II][JJ+1]; . . .
    for (k=0, i=II, j=JJ; k<N; k++) {
      T00 = T00+A[i][k]*B[k][j];
      T10 = T10+A[i+1][k]*B[k][j];
      T01 = T01+A[i][k]*B[k][j+1];
      T11 = T11+A[i+1][k]*B[k][j+1];
    }
    C[II][JJ]=T00;C[II+1][JJ]=T10; . . .
  }
```

TLB misses!!!

Tiling

- Example: Matrix-matrix multiplication, tiling for the cache level

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



Tiling

- Example: Matrix-matrix multiplication, tiling for the cache level

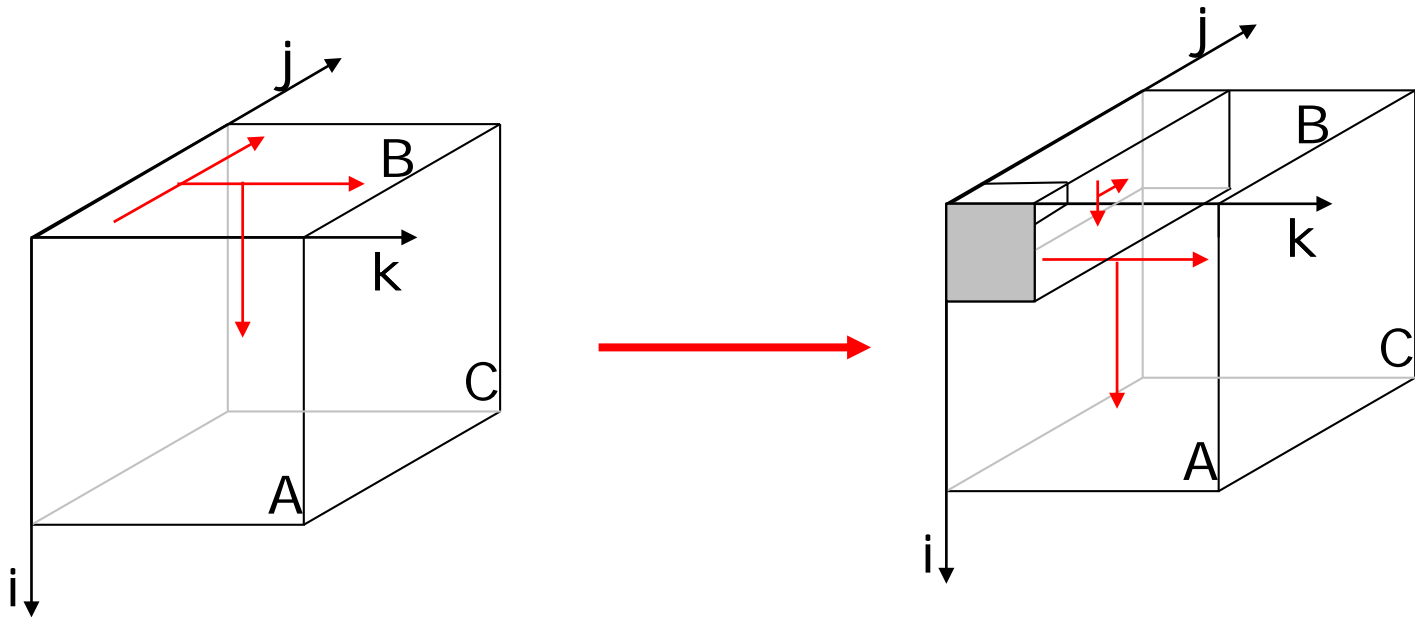
```
for (II=0; II<=N-1; II+=BII)
  for (JJ=0; JJ<=N-1; JJ+=BJJ)
    for (k=0; k<=N-1; k++)
      for (i=II; i<=min(N-1,BII-1); i++)
        for (j=JJ; j<=min(N-1,BJJ-1); j++)
          C[i][j] = C[i][j]+A[i][k]*B[k][j];
```

TLB misses!

Tiling

- Example: Matrix-matrix multiplication, tiling for the cache and register levels

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



Loop Order: III KKK JJ II K I J

Tiling

- Example: Matrix-matrix multiplication, tiling for the cache and register levels

```
for (III=0; III<N; III+=BIII)
  for (KKK=0; KKK<N; KKK+=BKKK)
    for (JJ=0; JJ<N; JJ+=2)
      for (II=III; II<MIN(N, III+BIII-1); II+=2){
        T00=C[II][JJ];T01=C[II][JJ+1]; . . .
        for (k=KKK, i=II, j=JJ; k<MIN(N,KKK+BKKK-1; k++) {
          T00 = T00+A[i][k]*B[k][j];
          T10 = T10+A[i+1][k]*B[k][j];
          T01 = T01+A[i][k]*B[k][j+1];
          T11 = T11+A[i+1][k]*B[k][j+1];
        }
        C[II][JJ]=T00;C[II+1][JJ]=T10; . . .
      }
}
```

Tiling

- Some basic references about Tiling:
 - Juan J. Navarro, Toni Juan and Tomás Lang. *MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations*. In *International Conference on Supercomputing (ICS'94)*, pp. 354-363, July 1994.
 - R.Schreiber and J.Dongarra. Automatic blocking of nested loops. Technical Report, RIACS, NASA Ames Research Center and Oak Ridge National Laboratory, May 1990.
 - M. Wolf. Improving locality and parallelism in nested loops. Ph.D. Thesis Stanford University, Aug 1992.
 - M. Wolfe. More iteration space tiling. *International Conference on Supercomputing (ICS'89)*, Jul 1989.

Basic Transformations

- Unimodular Transformations
 - Loop Interchange/Loop Permutation
 - Loop reversal
 - Loop Skewing
- Loop Fusion
- Loop Distribution
- Scalar Replacement
- Unroll & Jam
- Tiling
- Combination of transformations

Combination of transformations

- M. Wolf, D. Maydan, and D. Chen. *Combining loop transformations considering caches and scheduling*. MICRO 1996.
- S. Carr. Combining optimization for cache and Instruction-Level Parallelism. PACT 1996
- D. Gannon, W. Jalby and K. Gallivan. Strategies for cache and local memory management by global program transformations. ICS 1987
- M. Kandemir, J. Ramnujam and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. ICS 1997
- K. McKinley, S. Carr and C-W Tseng. Improving locality with loop transformations. TOPLAS 1996
- N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. TPDS 1997

Combination of transformations

- R. Saavedra et al. The combined effectiveness of unimodular transformations, tiling and software prefetching. IPPS 1996
- M. Wolf and M. Lam. A data locality optimizing algorithm. PLDI 1994
- T. Mowry, M. Lam and A. Gupta. Design and Evaluation of a compiler algorithm for prefetching. ASPLOS 1992