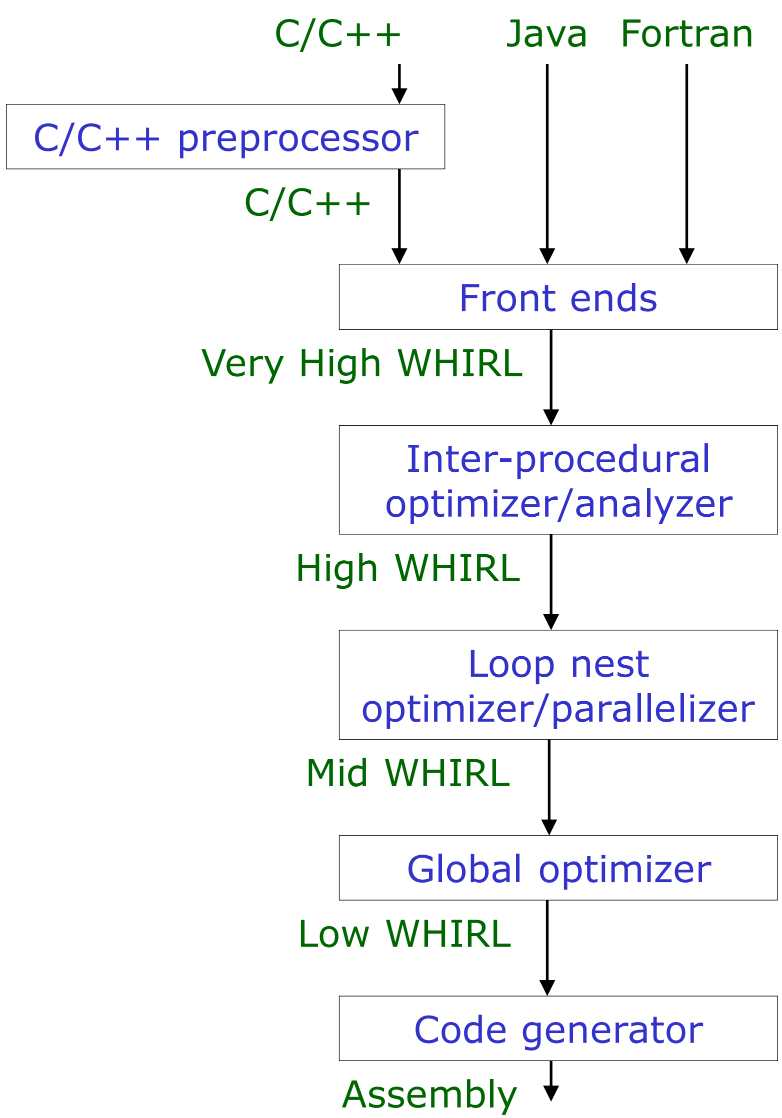
Memory Hierarchy Optimizations

Compilers for High Performance Architectures
(https://raco.fib.upc.es)

Josep Llosa, José Ramón Herrero, Marc González

Structure of a real compiler: ORC



Front end
Traditional compilers class
(parsing, analysis, etc…)
NOT this course topic

Middle end

(machine independent high-level optimizer)

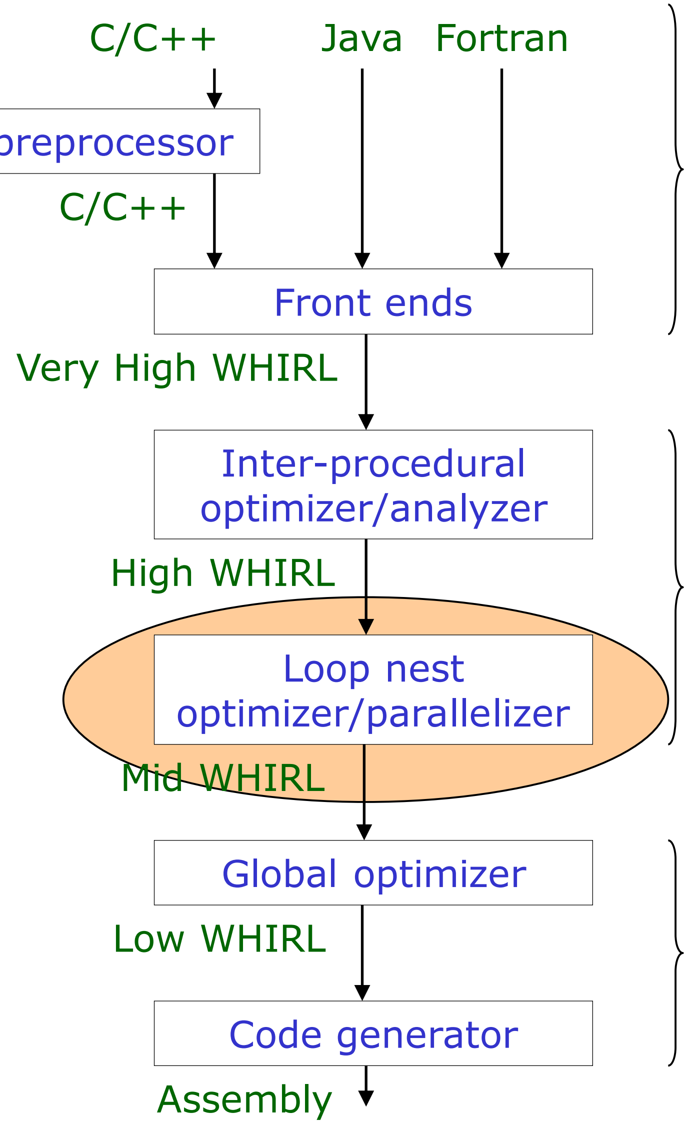
Covered in topics:
3) Memory optimizations
4) TLP optimizations

Back end

(machine dependent optimizations)

Covered in topic:
2) ILP optimizations

Structure of a real compiler: ORC



Front end
Traditional compilers class (parsing, analysis, etc…)
NOT this course topic

Middle end

(machine independent high-level optimizer)

Covered in topics:
3) Memory optimizations
4) TLP optimizations

Back end

(machine dependent optimizations)

Covered in topic:
2) ILP optimizations

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 (José Ramón Herrero)**
   - **Basic Concepts** **Acknowledgement: Marta Jiménez**
   - **Basic transformations**
   - **Loop vectorization**
4. Thread Level Parallelism Optimizations (Marc Gonzlez)
   - Thread level Parallelism
   - Analysis and detection of parallelism
   - Programming models
   - Parallel execution
   - Memory models

Basic Concepts

- Motivating Example
- Iteration Space
- Data Space and Affine Array Indexes
- Data Reuse
- Data Dependences

Motivating example

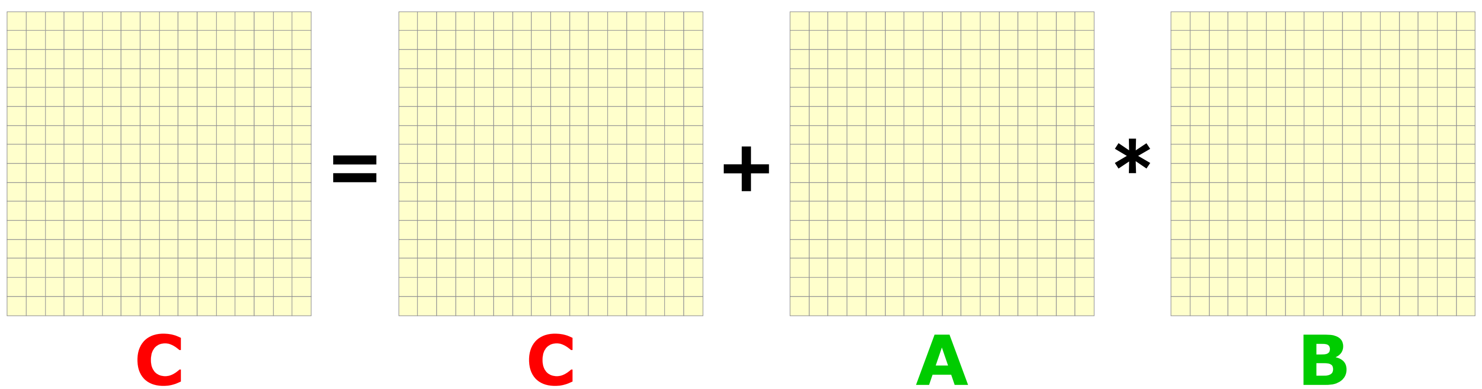
- Matrix Multiply

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

Motivating example

- Matrix Multiply

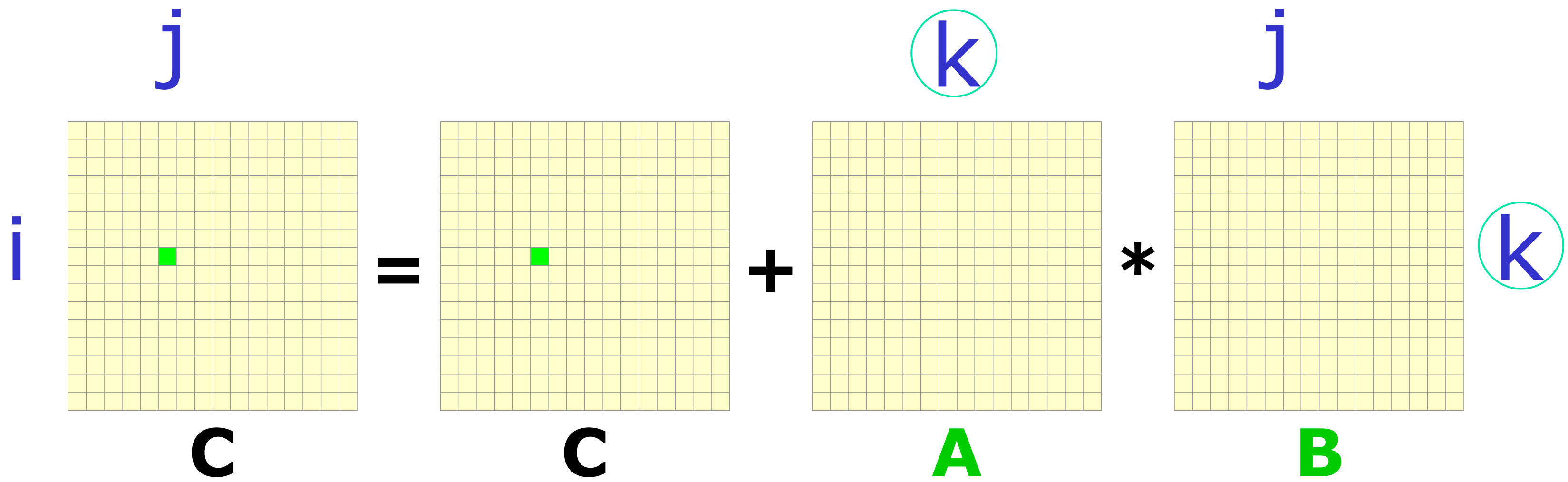
```
float A[N][N], B[N][N], C[N][N];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```



Motivating example

- Matrix Multiply

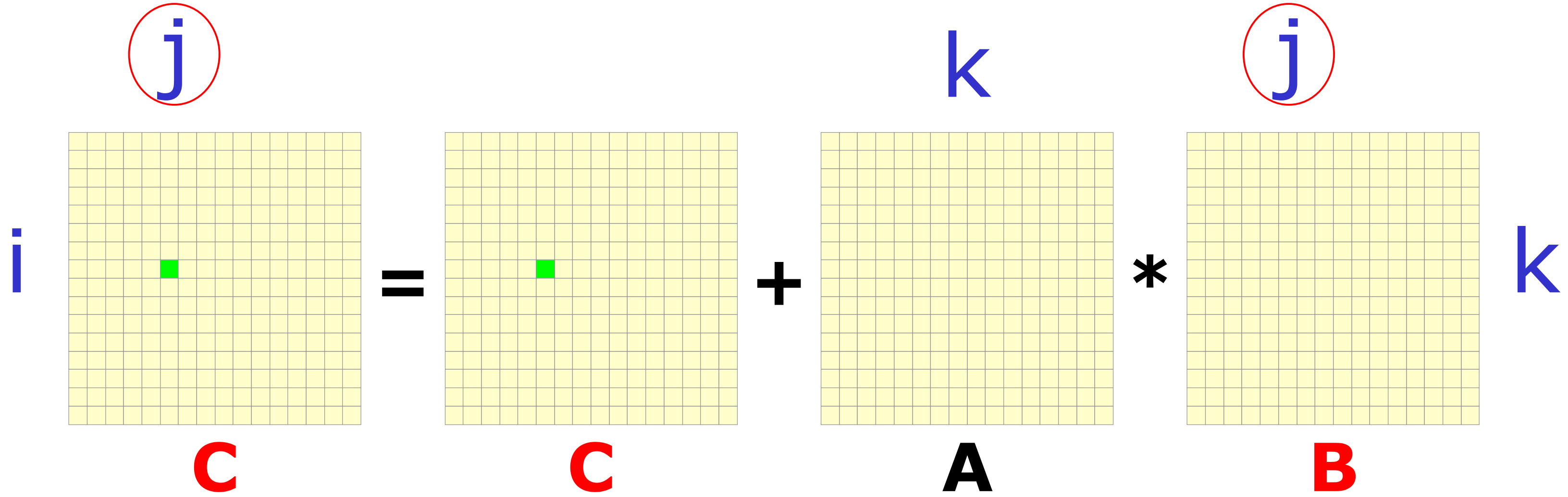
```
float A[N][N], B[N][N], C[N][N];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```



Motivating example

- Matrix Multiply

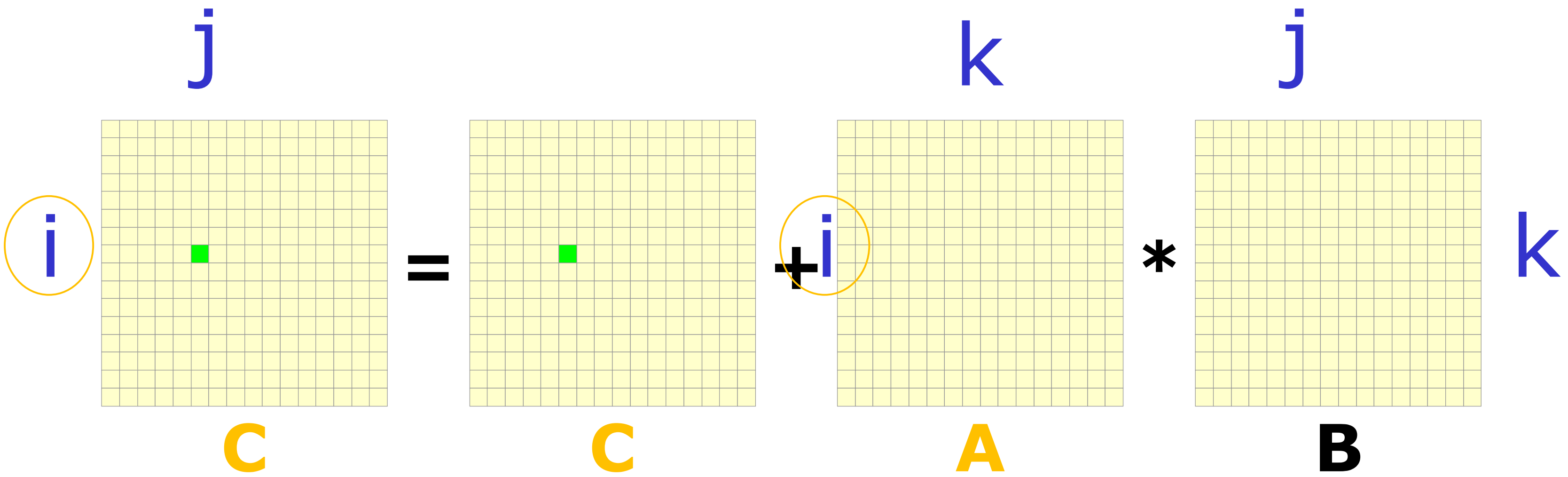
```
float A[N][N], B[N][N], C[N][N];
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
```



Motivating example

- Matrix Multiply

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

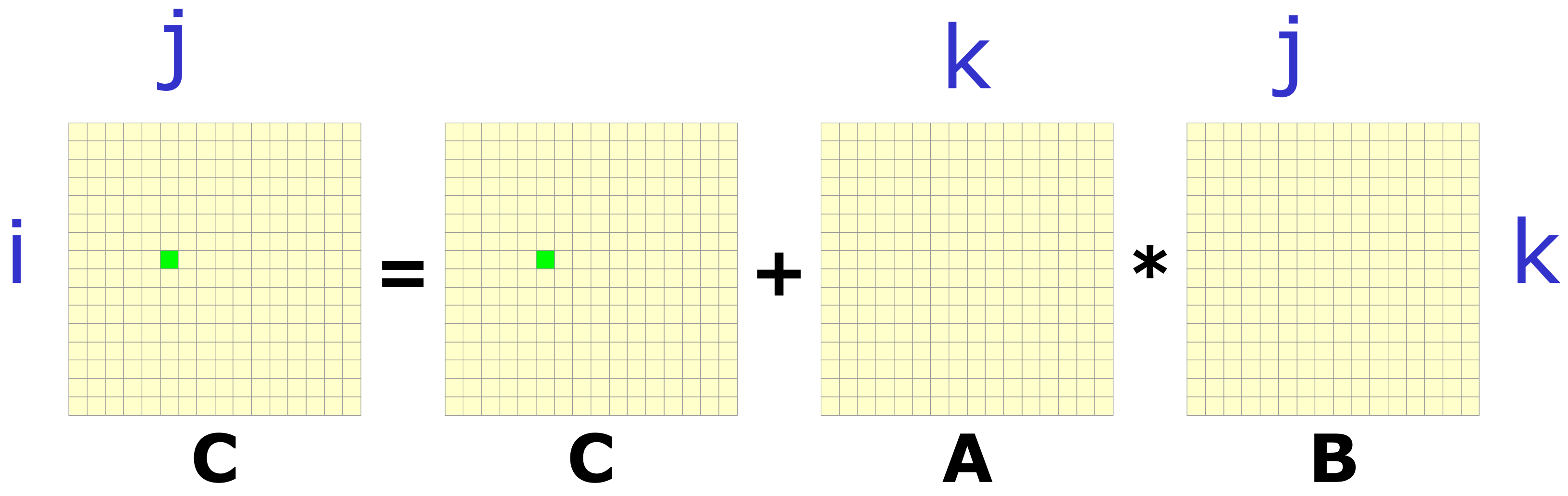


Motivating example

- Matrix Multiply

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

IJK form

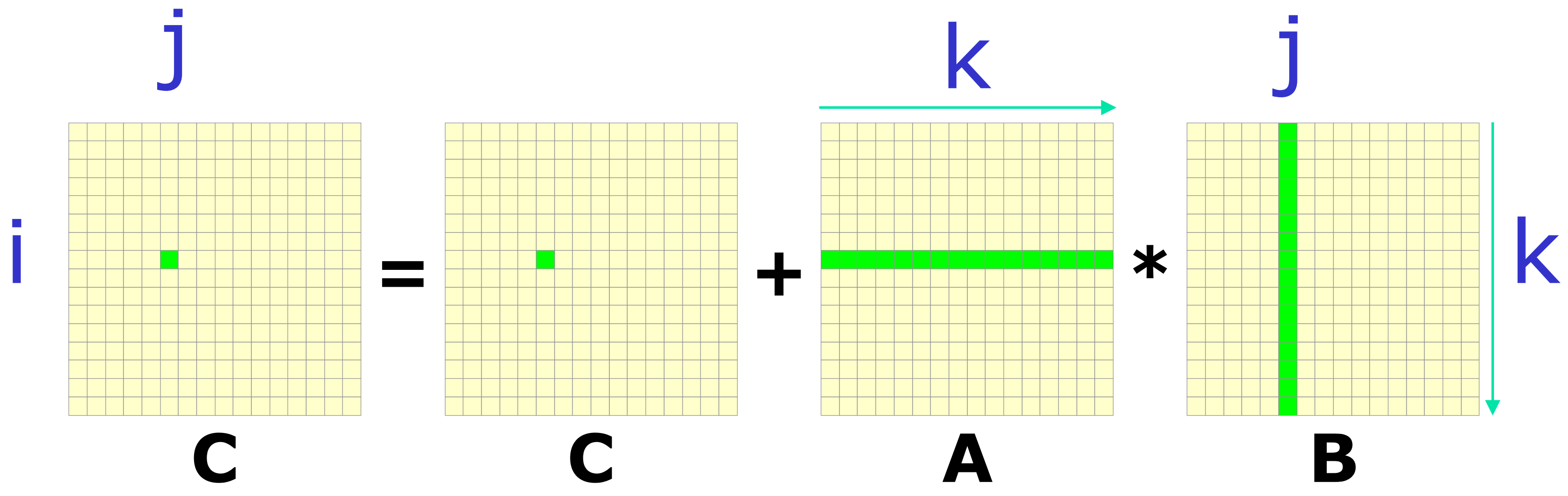


Motivating example

- Matrix Multiply

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

IJK form



Motivating example

- Matrix Multiply

IJK form

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

JKI form

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

KIJ form

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

Motivating example

- Matrix Multiply

IJK form

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

JKI form

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

KIJ form

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

In all 3 forms:

$3N^2$ memory locations

$4N^3$ memory access (75% are reads and 25% are writes)

$2N^3$ operations

Motivating Example

- Performance of different Matrix Multiply forms
- Execution time & MFLOPs for different matrix sizes (N)

Exec time (seg)	N=128	N=256	N=512	N=1024
IJK	0,02	0,18	1,63	57,62
JKI	0,02	0,22	2,93	156,92
KIJ	0,01	0,12	1,12	17,7

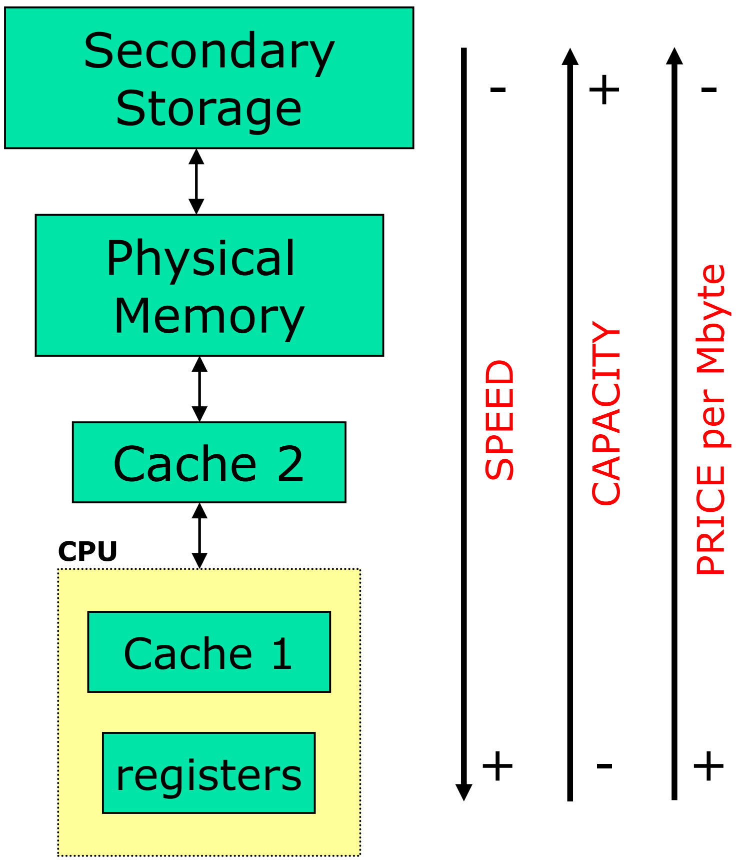
MFLOPs	N=128	N=256	N=512	N=1024
IJK	209,7	186,4	164,7	37,3
JKI	209,7	152,5	91,6	13,7
KIJ	419,4	279,6	239,7	121,3

- All 3 forms were executed under the same conditions on a Pentium platform (up to TFLOPS nowadays)
- Performance decreases as matrix size increases
- For N=1024, the best form is almost **10 times** faster than the worst

¿What produces this performance difference?

Motivating Example

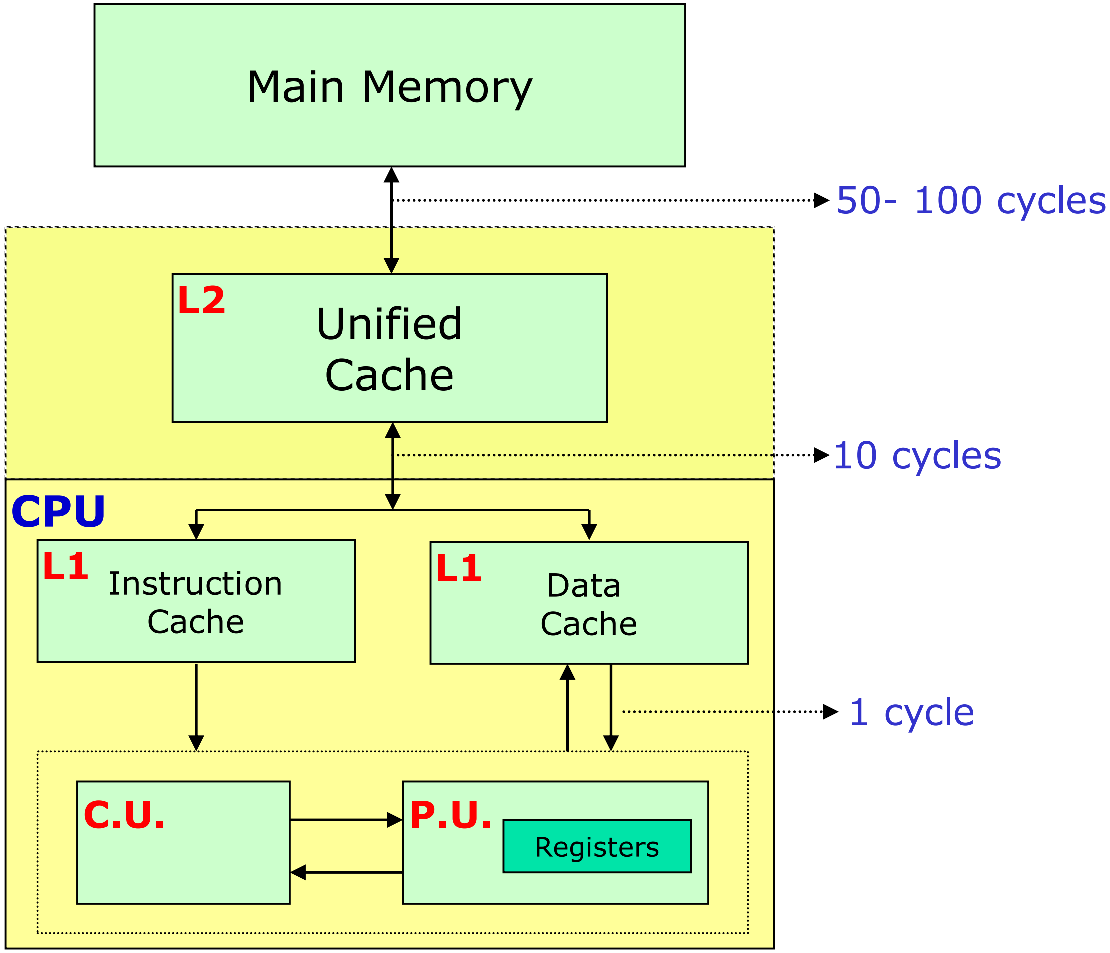
- The Memory Hierarchy



- The average memory access time of a program is reduced if most of its accesses are satisfied by the faster levels.
- Using registers effectively is probably the single most important problem in optimizing a program.
- Memory hierarchy depends on the data locality properties of programs for their effectiveness.

Motivating Example

- The performance difference between the three different matrix-multiply forms is caused by the memory hierarchy



Basic Concepts:

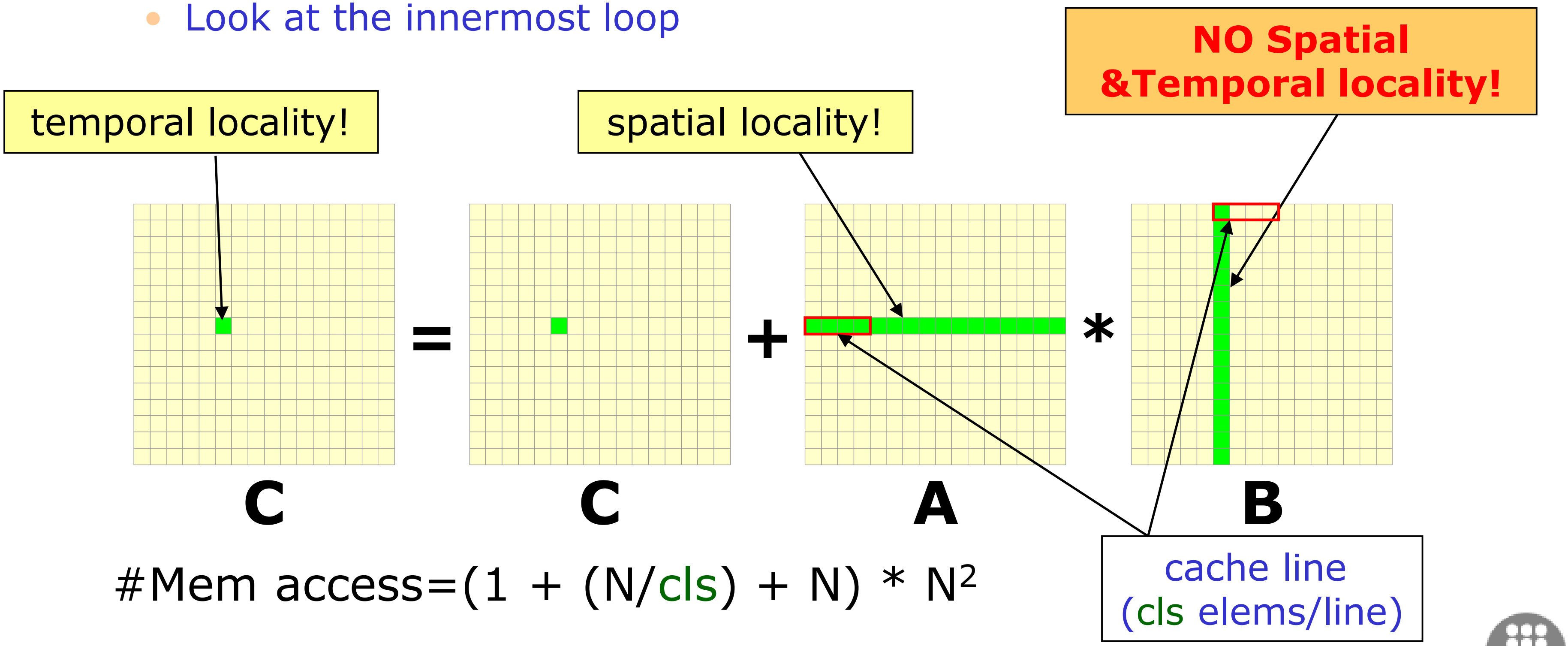
- Temporal locality

- Spatial locality

IJK-Form

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

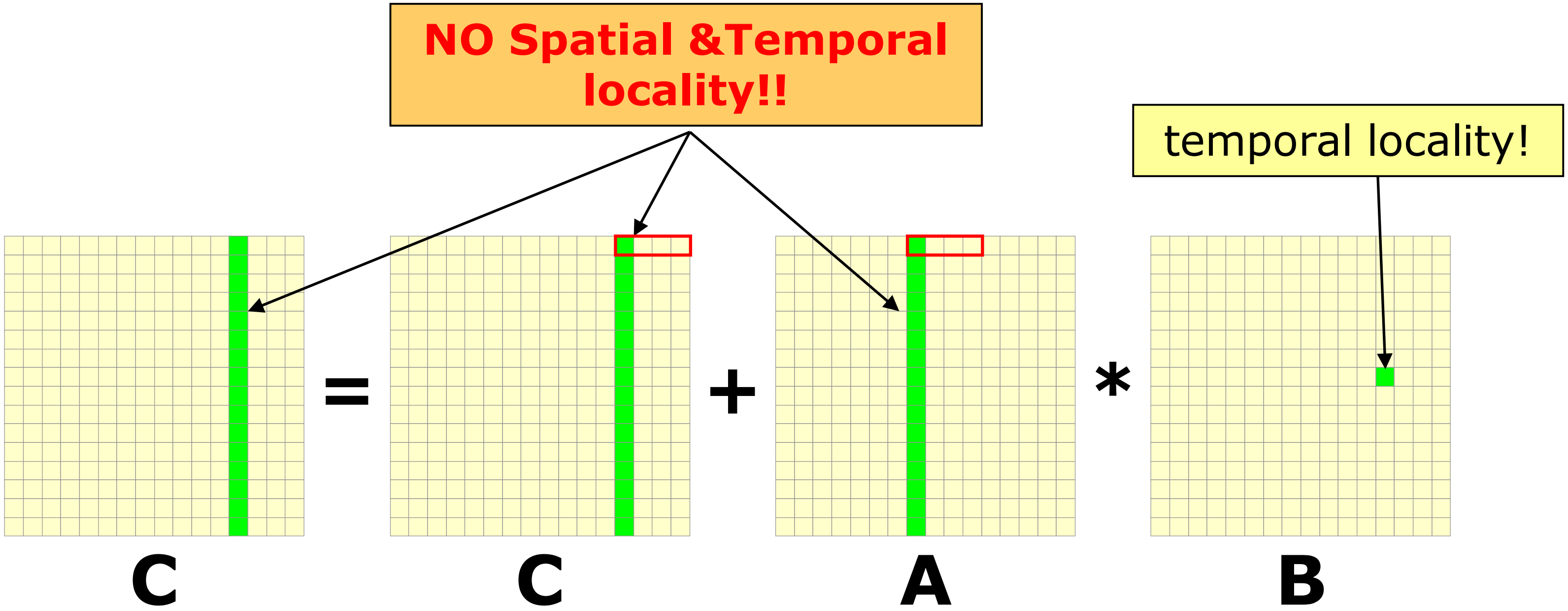
- ¿How are matrices traversed?
  - Look at the innermost loop



#Mem access=(1 + (N/cls) + N) * N^2

JKI-Form

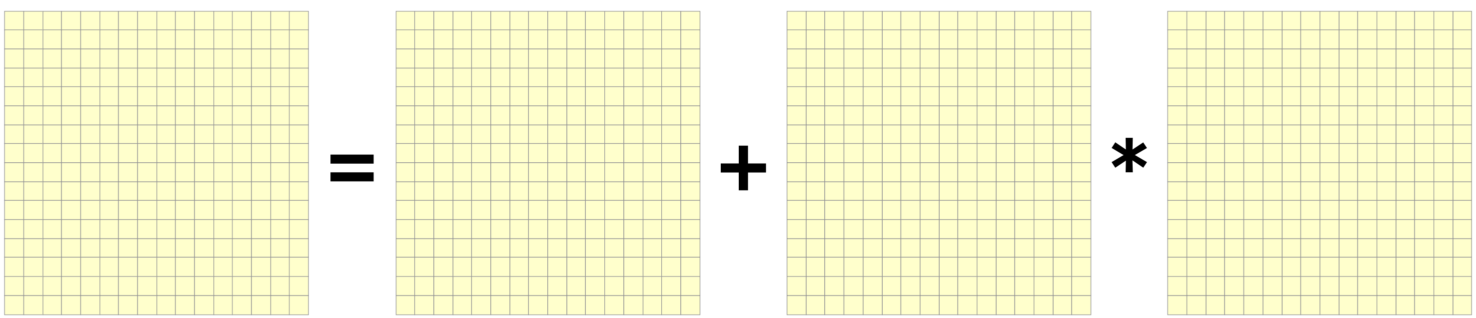
```
float A[N][N], B[N][N], C[N][N];
for (j=0; j<N; j++)
  for (k=0; k<N; k++)
    for (i=0; i<N; i++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```



#Mem access=(N + N + 1) * N^2

KIJ-Form

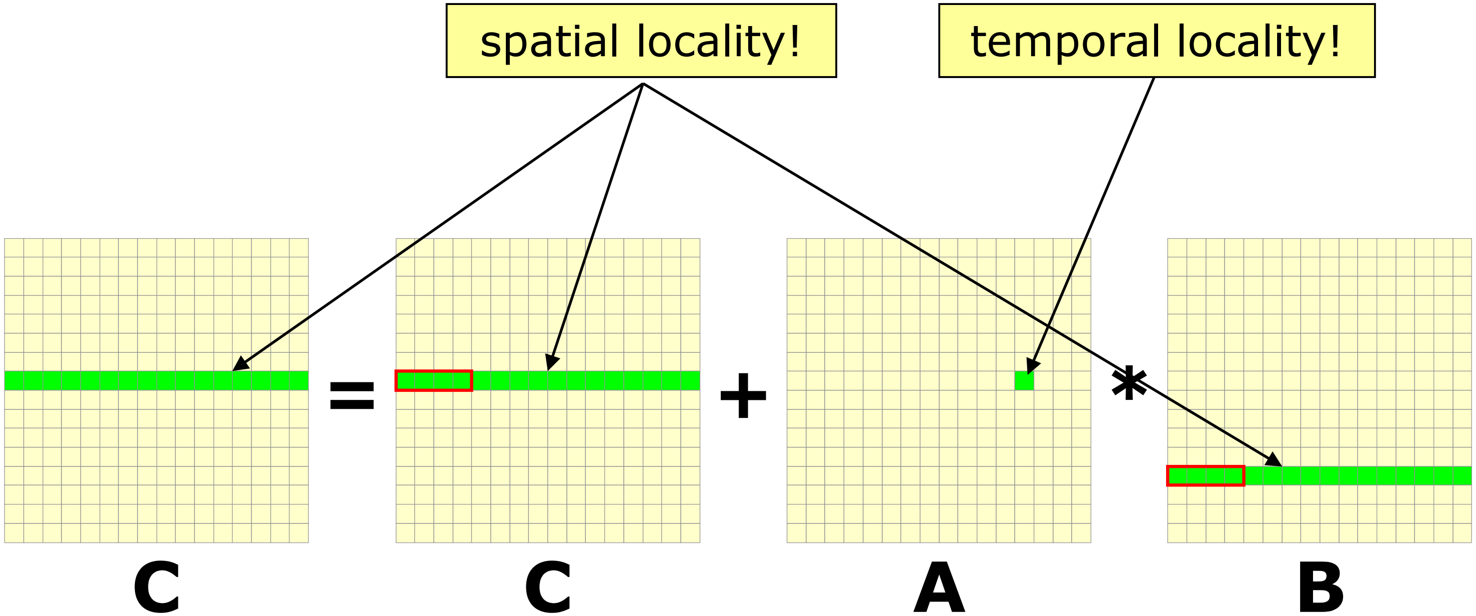
```
float A[N][N], B[N][N], C[N][N];
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```



C C A B

KIJ-Form

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



#Mem access=((N/cls) + 1 + (N/cls)) * N^2

An Example

- Now results can be explained

MFLOPs	N=128	N=256	N=512	N=1024	
IJK	209,7	186,4	164,7	37,3	→ Partial spatial locality
JKI	209,7	152,5	91,6	13,7	→ No spatial locality
KIJ	419,4	279,6	239,7	121,3	→ Spatial locality

- Can the compiler automatically choose the optimal loop permutation?
  - K. Kennedy and K. S. McKinley. *Optimizing for Parallelism and Data Locality*. 1992 ACM International Conference on Supercomputing (ICS 1992).
  - S. Carr, K. S. McKinley, and C. Tseng. *Compiler Optimizations for Improving Data Locality*. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994).

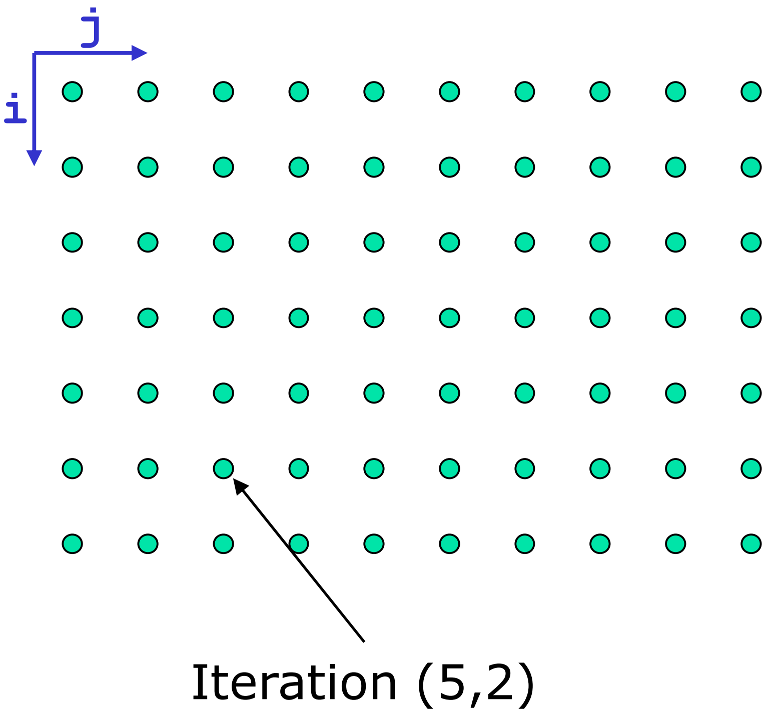
Basic Concepts

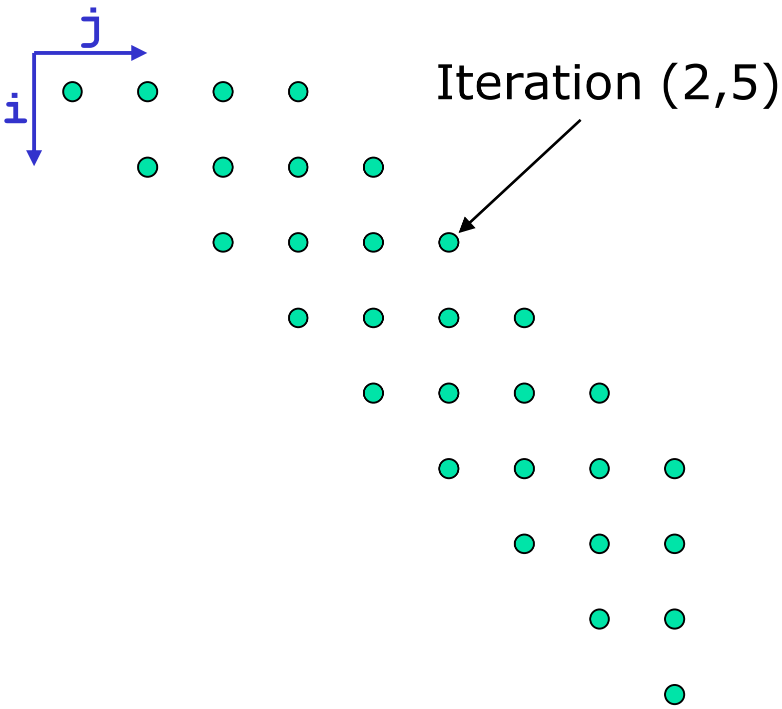
- An example
- Iteration Space
- Data Space and Affine Array Indexes
- Data Reuse
- Data Dependences

Iteration Space

```
for (i=0; i<=6; i++)
  for (j=0; j<=9; j++)
      . . .
```

```
for (i=0; i<=8; i++)
  for (j=i; j<=min(i+3,8); j++)
      . . .
```





IS: set of combinations of loop indices

Iteration Space

- IS representation of a d-deep loop

$\{I \text{ in } Z^d \mid B*I + b \geq 0\}$

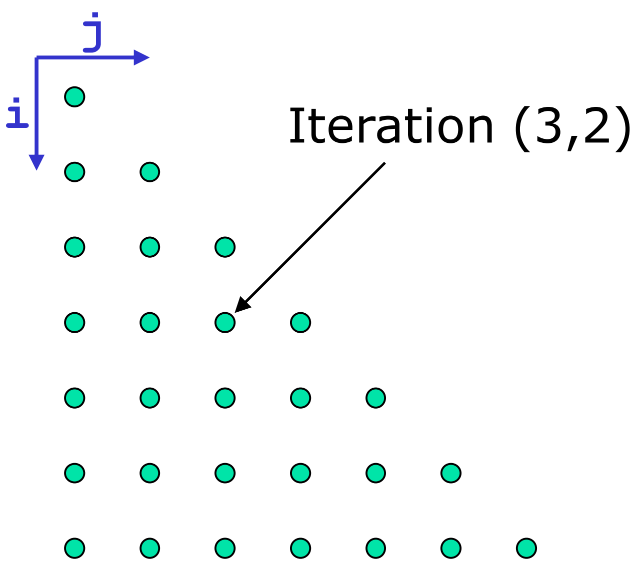
```
for (i=0; i<=6; i++)
  for (j=0; j<=i; j++)
      . . .
```

$i \geq 0$

$i \leq 6 \rightarrow -i + 6 \geq 0$

$j \geq 0$

$j \leq i \rightarrow i - j \geq 0$



$u*i + v*j + w \geq 0$

$B * I + b \geq 0$

Total number of Lower & Upper bounds

$$\begin{pmatrix}1 & 0\\ -1 & 0\\ 0 & 1\\ 1 & -1\end{pmatrix} * \begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}0\\ 6\\ 0\\ 0\end{pmatrix} \geq \begin{pmatrix}0\\ 0\\ 0\\ 0\end{pmatrix}$$

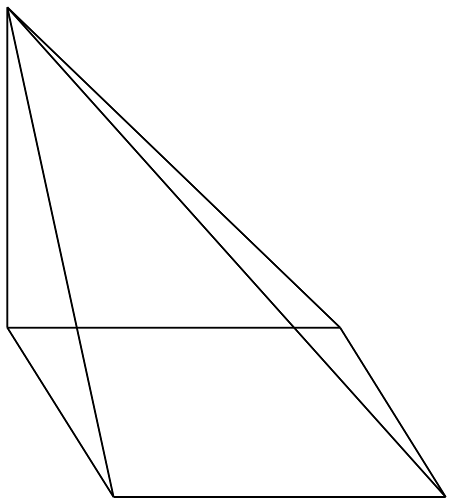
(u, v) **(w)**

d=Depth of loop nest

Iteration Space

- Exercise: how is this IS mathematically represented?

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

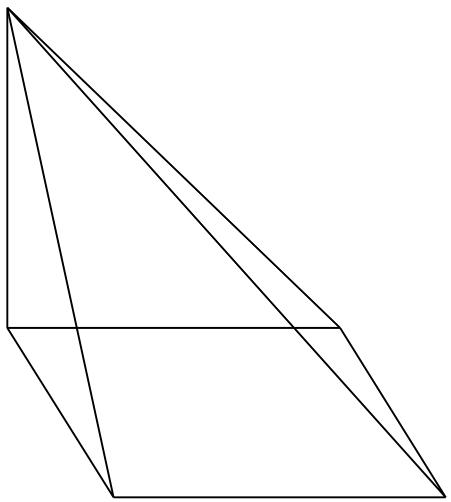


Iteration Space

- Exercise: how is this IS mathematically represented?

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

$$i \geq 0$$
$$i \leq N\text{-}1 \rightarrow -i + N\text{-}1 \geq 0$$
$$j \geq 0$$
$$j \leq N\text{-}1 \rightarrow -j + N\text{-}1 \geq 0$$
$$k \geq 0$$
$$k \leq i \rightarrow i - k \geq 0$$
$$k \leq j \rightarrow j - k \geq 0$$



$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ N-1 \\ 0 \\ N-1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

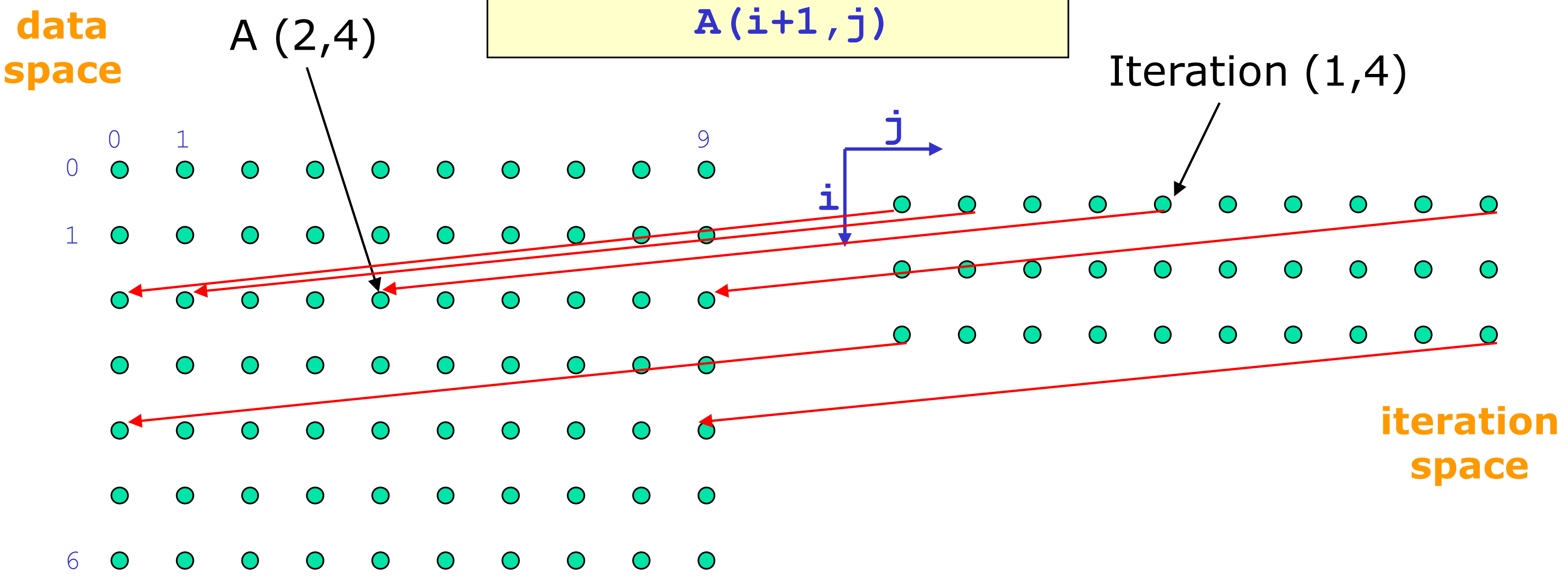
Basic Concepts

- An example
- Iteration Space
- Data Space and Affine Array Indexes
- Data Reuse
- Data Dependences

Data Space

- The data space (DS) is given directly by the array declaration

```
float A[7][10];
for (i=1; i<=3; i++)
   for (j=0; j<=9; j++)
         A(i+1,j)
```



- Each array access in the code specifies a mapping from an iteration in the IS to an array element in the DS

Affine Array Indexes

- Affine:
  - From Latin *affinis* ("connected with")
  - Allowing for or preserving parallel relationships.
  - Transformation which maps
    - Parallel lines to parallel lines
    - Finite points to finite points
- The access function is affine if it involves multiplying the loop index variables by constants and adding constants
  - Affine array accesses: `A(2*i+1, 3*j-10)`, `A(3*n, n-j)`
  - Nonaffine array accesses: `A(i*j,j*n)`

Affine Array Indexes

- Each array access in a d-deep loop nest is represented as:

$$\{I \text{ in } Z^d \mid F*I + f\}$$

F and f represent the function(s) of the loop-index variables that produce the array index(es) used in the various dimensions of the array access

`F` is `nxd`, being `n` the array dimensions and `d` the depth of the loop nest.

`I` is a vector with as many elements as loops (`d`)

`f` is a vector with as many elements as array dimensions (`n`)

Affine Array Indexes

- Examples

$$X[i-1] \rightarrow [1 \quad 0] * \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$$

$$Y[j, j+1] \rightarrow \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$Z[1, i, 2*i+j] \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$Y[1,2] \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- Affine functions provide a special mapping from the IS to the DS, making it easy to determine which iterations map to the same data or same cache line

Affine Array Indexes

- Exercise

  For each of the following array accesses, give the vector f and the matrix F that describe them. Assume that the surrounding loops are `i,j` and `k`.

  a) `X[2*i+3, 2*j-i]`

  b) `Y[i-j, j-k, k-i]`

  c) `Z[3,2*j, k-2*i+1]`

Affine Array Indexes

- Exercise

  For each of the following array accesses, give the vector f and the matrix F that describe them. Assume that the surrounding loops are i,j and k.

a) X[2*i+3, 2*j-i]

$$\begin{bmatrix} 2 & 0 & 0 \\ -1 & 2 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

b) Y[i-j, j-k, k-i]

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

c) Z[3,2*j, k-2*i+1]

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ -2 & 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$$

Basic Concepts

- An example
- Iteration Space
- Data Space and Affine Array Indexes
- Data Reuse
- Data Dependences

Data Reuse

- Types of reuse
  - Self-Temporal Reuse: same exact location referenced by same static access
  - Self-Spatial Reuse: same cache line referenced by same static access
  - Group-Temp Reuse: same exact location referenced by different static access
  - Group-Spatial Reuse: same cache line referenced by different static access

```
float Z[n];
for (i = 0; i < n; i++)
  for (j = 0; j < n-2; j++)
       Z[j+1] = (Z[j]+Z[j+1] + Z[j+2])/3;
```

  - $4n^2$ accesses in this code
  - If reuse exploited, we only need to bring n/cls cache lines

Data Reuse

- Self-Temporal Reuse: same exact location referenced by same static access (in different iterations)
- A static access can be represented as $A(F \cdot I+f)$
- Let iterations I_1 and I_2 refer to the same array element, then $F \cdot I_1+f=F \cdot I_2+f$, and therefore $F \cdot (I_1-I_2)=0$.
- In linear algebra, the set of all solutions to the equation $F \cdot r=0$ is called the null space of F
- If F is fully ranked, then its null space is the null vector
- We say that there is self-temporal reuse in direction r when $F \cdot r=0$ and $r \in$ {null space of F }

Data Reuse

- Examples of Self-Temporal Reuse in a 2-deep loop nest

$$Z[1,i,2*i+j] \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Basis of Null Space of F
{}

$$X[i-1] \rightarrow \begin{bmatrix} 1 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$$

Basis of Null Space of F
{(0,1)}

$$Y[j,j+1] \rightarrow \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Basis of Null Space of F
{(1,0)}

$$Y[1,2] \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Basis of Null Space of F
{(1,0),(0,1)}

Data Reuse

- Self-Spatial Reuse: same cache line referenced by same static access (in different iterations)
- Row major order vs. Column major order
  - X[i,j] and X[i,j+1] contiguous in row major order
  - X[i,j] and X[i+1,j] contiguous in column major order
- Two array elements share the same cache line iff they share the same row in a 2-dimensional array.
- In an array of d dimensions, array elements share the same cache line if they differ only in the last dimension.
- Drop the last row of F to construct F_s and compute the basis of the null space of F_s
- We say that there is self-spatial reuse in direction r when $F_s \cdot r=0$ and $r \in \{\text{null space of } F_s - \text{null space of } F\}$

Data Reuse

- Example of Self-Spatial Reuse: array access in a 2-deep loop nest

$$Z[1,i,2*i+j] \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$F = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow F_S = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

Basis of Null Space of F_s

{(0,1)}

- Z[1, i, 2*i+j] has self-spatial reuse in direction j

- What about Z[1, i, 2*i+50*j] ?

Data Reuse

- Group-Temporal Reuse: same exact location referenced by different static access

- Given two dynamic accesses $F \cdot I_1 + f_1$ and $F \cdot I_2 + f_2$, reuse of the same data requires that $F \cdot I_1 + f_1 = F \cdot I_2 + f_2$, and therefore

$$F \cdot (I_1 - I_2) = (f_2 - f_1)$$

- Let v be one solution to this equation. Then if w is any vector in the null space of F, $w+v$ is also a solution and those are all the solutions to the equation

- Analogously for group-spatial reuse

Example

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
       Z[i,j] = Z[i-1,j];
```

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow F_S = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

- Z[i,j] and Z[i-1,j] do not have self-temporal reuse, but they have self-spatial reuse

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \Rightarrow \begin{matrix} j_1 = j_2 \\ i_2 = i_1 + 1 \end{matrix} \Rightarrow v = (-1,0)$$

- There is group-temporal reuse along the i-axis of the IS between the two accesses Z[i,j] and Z[i-1,j]

Data Reuse

- Exercise: Compute the type of data reuse in matrix-matrix multiplication for each array access

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

Data Reuse

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

- If we focus on C[i][j], then we have: $F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ $F_s = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
- and null space of F = {(0,0,1)} and null space of F_s = {(0,1,0),(0,0,1)}
- Therefore,
  - C[i][j] has self-temporal reuse in direction k
  - C[i][j] has self-spatial reuse in direction j
  - A[i][k] has self-temporal reuse in direction j
  - A[i][k] has self-spatial reuse in direction k
  - B[k][j] has self-temporal reuse in direction i
  - B[k][j] has self-spatial reuse in direction j

Basic Concepts

- An example
- Iteration Space
- Data Space and Affine Array Indexes
- Data Reuse
- Data Dependences

Data Dependences

- Operation can be reordered *iff* the reordering does not change the program´s output
  - The compiler check that operations on any memory locations are done in the same order in the original and modified programs

- We focus on array accesses

- Two accesses are data dependent if:
  - They refer to the same memory location
  - At least one of them is a write

- The relative execution ordering between every pair of data-dependent operations in the original program must be preserved in the new optimized program

Data Dependences

- Types of dependences
  - True dependence (RAW)

```
S1: A = B + C
S2: D = A * 3
```

$$\begin{tikzcd} S1 \arrow[d] \\ S2 \end{tikzcd}$$

$S1\ \delta\ S2$

  - Antidependence (WAR)

```
R1: X = Y / 25
R2: Y = Z + 1
```

$$\begin{tikzcd} R1 \arrow[d, "|" marking] \\ R2 \end{tikzcd}$$

$R1\ \bar{\delta}\ R2$

  - Output Dependence (WAW)

```
T1: M = N / Q
T2: M = P - F
```

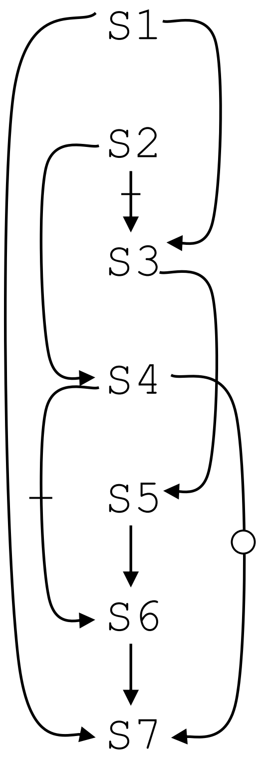
$$\begin{tikzcd} T1 \arrow[d, "\circ" marking] \\ T2 \end{tikzcd}$$

$T1\ \delta^{o}\ T2$

Data Dependences

- Dependence Graph

```
S1: A = 6
S2: B = C * 6
S3: C = A + 3
S4: D = B + F
S5: E = C
S6: F = E + C
S7: D = F + A
```



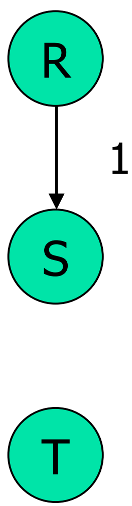
- In memory hierarchy optimizations, we need to compute data dependence of array accesses

Data Dependences

```
for (i=2; i<200; i++){
 R: A[i]=B[i]+C [i]
 S: B[i+2]=A[i-1]+C[i-1]
 T: A[i+1]=B[2*i+3]+1
}
```

unrolling

```
 (i=2)
A(2)=B(2)+C(2)
B(4)=A(1)+C(1)
A(3)=B(7)+1

 (i=3)
A(3)=B(3)+C(3)
B(5)=A(2)+C(2)
A(4)=B(9)+1

 (i=4)
A(4)=B(4)+C(4)
B(6)=A(3)+C(3)
A(5)=B(11)+1

 (i=5)
A(5)=B(5)+C(5)
B(7)=A(4)+C(4)
A(6)=B(13)+1
```

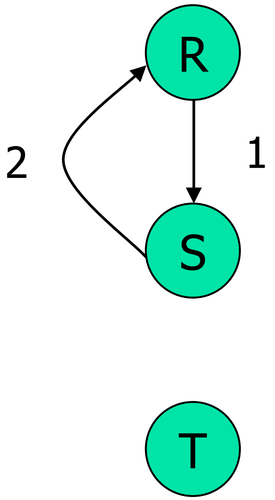


Data Dependences

```
for (i=2; i<200; i++){
 R: A[i]=B[i]+C [i]
 S: B[i+2]=A[i-1]+C[i-1]
 T: A[i+1]=B[2*i+3]+1
}
```

unrolling

```
(i=2)
A(2)=B(2)+C(2)
B(4)=A(1)+C(1)
A(3)=B(7)+1

(i=3)
A(3)=B(3)+C(3)
B(5)=A(2)+C(2)
A(4)=B(9)+1

(i=4)
A(4)=B(4)+C(4)
B(6)=A(3)+C(3)
A(5)=B(11)+1

(i=5)
A(5)=B(5)+C(5)
B(7)=A(4)+C(4)
A(6)=B(13)+1
```

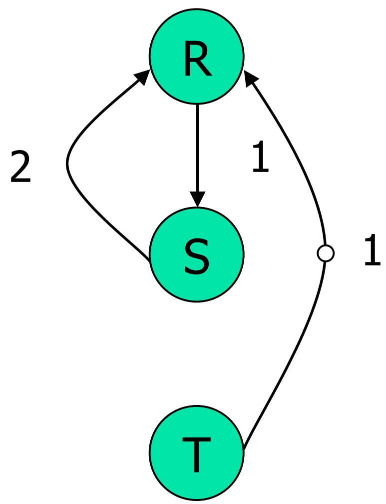


Data Dependences

```
for (i=2; i<200; i++){
 R: A[i]=B[i]+C [i]
 S: B[i+2]=A[i-1]+C[i-1]
 T: A[i+1]=B[2*i+3]+1
}
```

unrolling

```
(i=2)
A(2)=B(2)+C(2)
B(4)=A(1)+C(1)
A(3)=B(7)+1

(i=3)
A(3)=B(3)+C(3)
B(5)=A(2)+C(2)
A(4)=B(9)+1

(i=4)
A(4)=B(4)+C(4)
B(6)=A(3)+C(3)
A(5)=B(11)+1

(i=5)
A(5)=B(5)+C(5)
B(7)=A(4)+C(4)
A(6)=B(13)+1
```

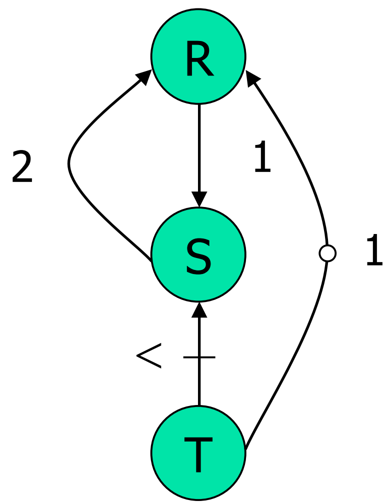


Data Dependences

```
for (i=2; i<200; i++){
 R: A[i]=B[i]+C [i]
 S: B[i+2]=A[i-1]+C[i-1]
 T: A[i+1]=B[2*i+3]+1
}
```

unrolling

```
(i=2)
A(2)=B(2)+C(2)
B(4)=A(1)+C(1)
A(3)=B(7)+1

(i=3)
A(3)=B(3)+C(3)
B(5)=A(2)+C(2)
A(4)=B(9)+1

(i=4)
A(4)=B(4)+C(4)
B(6)=A(3)+C(3)
A(5)=B(11)+1

(i=5)
A(5)=B(5)+C(5)
B(7)=A(4)+C(4)
A(6)=B(13)+1

(i=6)
A(6)=B(6)+C(6)
B(8)=A(5)+C(5)
A(7)=B(15)+1

(i=7)
A(7)=B(7)+C(7)
B(9)=A(6)+C(6)
A(8)=B(17)+1
```



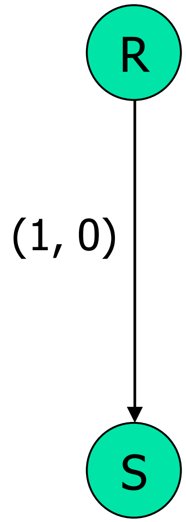
Range of distances	direction
[1,∞]	<
[-∞,-1]	>
[-∞, ∞]	*

Data Dependences

```
for (i=1; i<200; i++){
  for (j=1; j<200; j++) {
    R: A[i,j]=B[i]+C[j]
    S: B[i+2]=A[i-1,j]+C[i-1]
}
```

unrolling

```
(i=1, j=1)
A(1,1)=B(1)+C(1)
B(3)=A(0,1)+C(0)

(i=1, j=2)
A(1,2)=B(1)+C(2)
B(3)=A(0,2)+C(0)

(i=1, j=3)
A(1,3)=B(1)+C(3)
B(3)=A(0,3)+C(0)

. . .

(i=2, j=1)
A(2,1)=B(2)+C(1)
B(4)=A(1,1)+C(1)

(i=2, j=2)
A(2,2)=B(2)+C(2)
B(4)=A(1,2)+C(1)

. . .
(i=3, j=1)
A(3,1)=B(3)+C(1)
B(5)=A(2,1)+C(2)

(i=3, j=2)
A(3,2)=B(3)+C(2)
B(5)=A(2,2)+C(2)

. . .
```

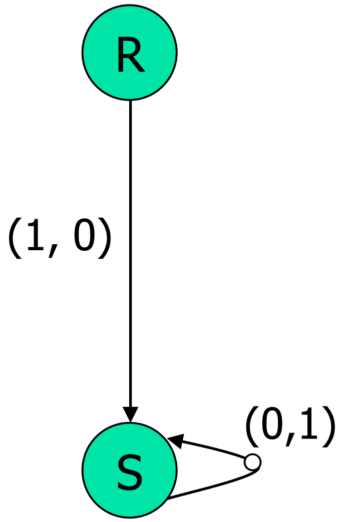


Data Dependences

```
for (i=1; i<200; i++){
  for (j=1; j<200; j++) {
    R: A[i,j]=B[i]+C[j]
    S: B[i+2]=A[i-1,j]+C[i-1]
}
```

unrolling

```
(i=1, j=1)
A(1,1)=B(1)+C(1)
B(3)=A(0,1)+C(0)

(i=1, j=2)
A(1,2)=B(1)+C(2)
B(3)=A(0,2)+C(0)

(i=1, j=3)
A(1,3)=B(1)+C(3)
B(3)=A(0,3)+C(0)

. . .

(i=2, j=1)
A(2,1)=B(2)+C(1)
B(4)=A(1,1)+C(1)

(i=2, j=2)
A(2,2)=B(2)+C(2)
B(4)=A(1,2)+C(1)

. . .
(i=3, j=1)
A(3,1)=B(3)+C(1)
B(5)=A(2,1)+C(2)

(i=3, j=2)
A(3,2)=B(3)+C(2)
B(5)=A(2,2)+C(2)

. . .
```

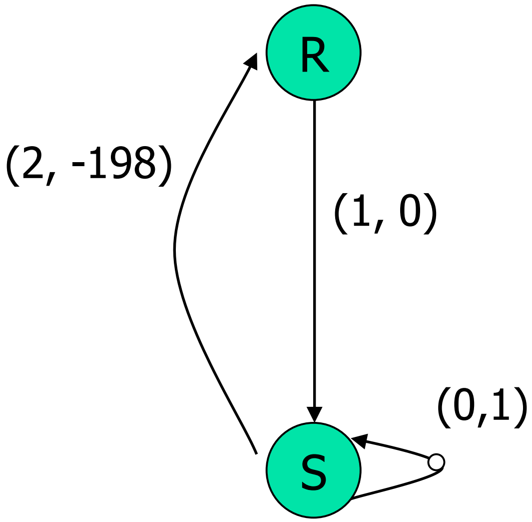


Data Dependences

```
for (i=1; i<200; i++){
  for (j=1; j<200; j++) {
    R: A[i,j]=B[i]+C[j]
    S: B[i+2]=A[i-1,j]+C[i-1]
}
```

unrolling

```
(i=1, j=1)
A(1,1)=B(1)+C(1)
B(3)=A(0,1)+C(0)

(i=1, j=2)
A(1,2)=B(1)+C(2)
B(3)=A(0,2)+C(0)

. . .

(i=1, j=199)
A(1,3)=B(1)+C(199)
B(3)=A(0,3)+C(0)

(i=2, j=1)
A(2,1)=B(2)+C(1)
B(4)=A(1,1)+C(1)

(i=2, j=2)
A(2,2)=B(2)+C(2)
B(4)=A(1,2)+C(1)

. . .
(i=3, j=1)
A(3,1)=B(3)+C(1)
B(5)=A(2,1)+C(2)

(i=3, j=2)
A(3,2)=B(3)+C(2)
B(5)=A(2,2)+C(2)

. . .
```



Data Dependences

- General Definition of Data Dependence of Array Accesses

```
for (i₁=Linf i₁<=Lsup; i₁++)
  for (i₂=Linf i₂<=Lsup; i₂++)
  …
     for (iₙ=Linf iₙ<=Lsup; iₙ++){

            R: . . . A[F₁*I+f₁]. . .
            S: . . . A[F₂*I+f₂]. . .
}
```

- There is a data dependence between R and S if:
  - At least one of the accesses is a write reference
  - There exist vector I_1 and I_2 in Z^d such that:
    - $\mathbf{F_1 * I_1 + f_1} = \mathbf{F_2 * I_2 + f_2}$
    - $\mathbf{B * I_1 + b} \geq 0$
    - $\mathbf{B * I_2 + b} \geq 0$
- Data dependence requires finding integer solutions that satisfy a set of linear inequality, that is integer linear programming

Data Dependece

- Data dependence analysis algorithm:

  1- Using the theory of linear Diophantine equations, check for the existence of integer solution to the equalities (GCD test):

  a) no integer solution →no data dependence

  b) otherwise, go to step 2

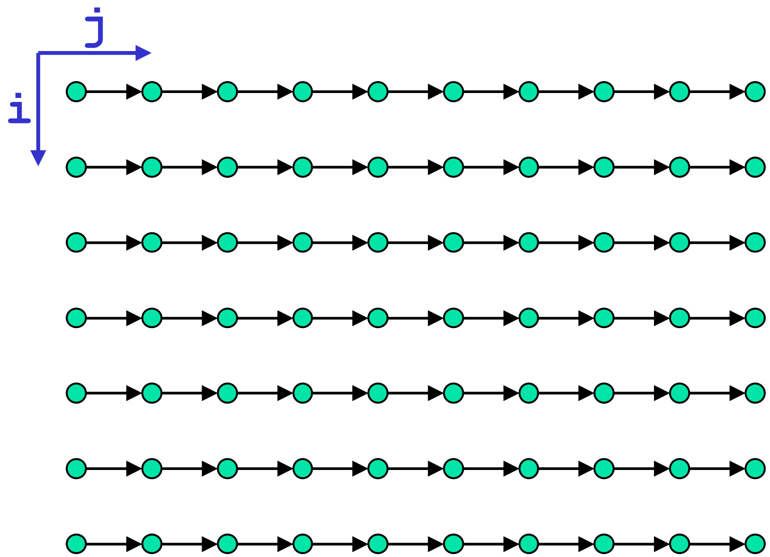
  2- Use a set of simple heuristics for solving integer linear problems (Acyclic Test, Independent-Variables Test, Loop-Residue Test, etc.)

  3- If heuristics do not work, we use a linear integer programming solver based on Fourier-Motzkin elimination

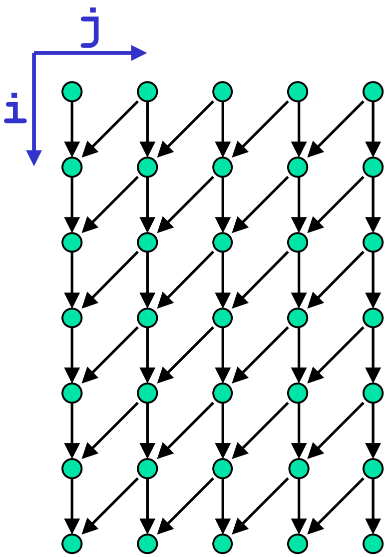
Iteration Space Dependence Graph

```
for (i=0; i<=6; i++)
  for (j=1; j<=10; j++)
    A[i,j] = F(A[i,j-1]);
```

```
for (i=1; i<=7; i++)
  for (j=1; j<=5; j++)
    A[i,j]=F(A[i-1,j],A[i-1,j+1]);
```



$d_1=(0,1)$



$d_1=(1,0)$

$d_2=(1,-1)$