Apps continued and Loop Transformations

Announcements

– Quiz 1 is on RamCT and is due Friday night

– HW1 is due Wednesday February 8th

Today

– Finishing discussion about scientific apps

  – What is their operational intensity?

  – Where is the data reuse?

  – Where is the parallelism?

– Starting Loop Transformations for Data Locality

  – Loop Permutation

  – Data dependences

  – Legality of Loop Permutation

Acknowledgement

– Some of these slides were originally created by Calvin Lin at UT, Austin.

1D Stencil Computation

Stencil Computations

– Computations operate over some mesh or grid

– Computation is modifying the value of something over time or as part of a relaxation to find steady state

– Each computation has some nearest neighbor data dependence pattern

– The coefficients multiplied by neighbor can be constant or variable

1D Stencil Computation version 1 <demo in class>

```
// assume A[0,i] initialized to some values
for (t=1; t<(T+1); t++) {
  for (i=1; i<(N-1); i++) {
    A[t,i] = 1/3 * (A[t-1,i-1] + A[t-1,i] + A[t-1,i+1];
  }
}
```

1D Stencil Computation (take 2)

1D Stencil Computation, version 2 <demo in class>

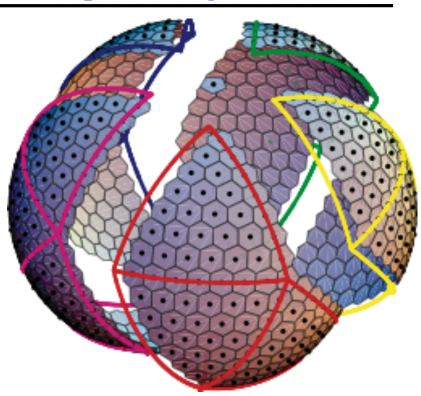
```
// assume A[i] initialized to some values
for (t=0; t<T; t++) {
  for (i=1; i<(N-1); i++) {
    A[i] = 1/3 * (A[i-1] + A[i] + A[i+1];
  }
}
```

Analysis

- Are version 1 and version 2 computing the same thing?
- What is the operational intensity of version 1 versus version 2?
- What parallelism is there in version 1 versus version 2?
- Where is the data reuse in version 1 versus version 2?

Jacobi in SWM code (Stencil Computation with Explicit Weights)

Source: David Randall's research group



```
do ksdm=1,nsdm
   do j=npad+1,ny-npad
      do i=npad+1,nx-npad
         work(i,j,:) =
             rw7(i, j, ksdm) * ( + rhs( i, j, :, ksdm)                &
                - l_weights( 1, i, j, ksdm) * xout(i-1, j  , :, ksdm)&
                - l_weights( 2, i, j, ksdm) * xout(i-1, j-1, :, ksdm)&
                - l_weights( 3, i, j, ksdm) * xout(i  , j-1, :, ksdm)&
                - l_weights( 4, i, j, ksdm) * xout(i+1, j  , :, ksdm)&
                - l_weights( 5, i, j, ksdm) * xout(i+1, j+1, :, ksdm)&
                - l_weights( 6, i, j, ksdm) * xout(i  , j+1, :, ksdm))
      enddo
   enddo
 enddo
```

Forward Substitution (Dense Matrix)

Given an NxN lower triangular matrix with unit diagonals and a n-vector b solve for the vector x in $Lx = b$

$$b_i = \sum_{j=1}^{N} L_{i,j} xj$$

How do we solve for x?

How do we turn this into a loop program?

Moldyn <draw iteration space>

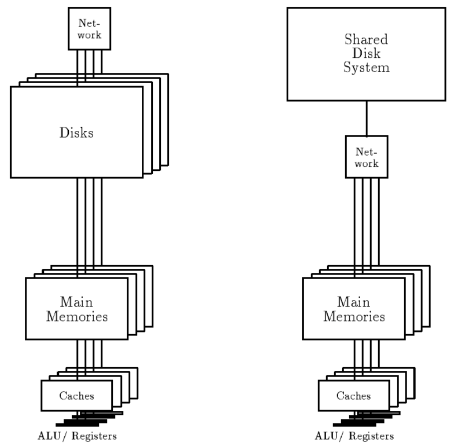
```
for (tstep=0;tstep<=n_tstep-1;tstep++) {
   …
   for (i=0;i<=n_moles-1;i++) {
     x(i) = x(i) + vhx(i) + fx(i);
     …
     if ( x(i) < 0.0 )  x(i) = x(i) + side ; …
     if ( x(i) > side ) x(i) = x(i) - side ; …

     vhx(i) = vhx(i) + fx(i); …
     fx(i)  = 0.0; …
   }
   for (ii=0;ii<=n_inter-1;ii++) {
     i = inter1(ii);  j = inter2(ii);
     fx(i) += … x(i)… x(j)…
     fx(j) += … x(i)… x(j)…
   }
   for (i=0;i<=n_moles-1;i++) {
     …
     vhx(i) = … fx(i) …; …
   }
}
```

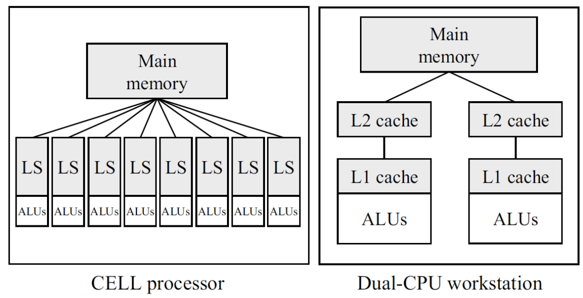
The Problem: Mapping programs to architectures

Goal: keep each core as busy as possible

Challenge: get the data to the core when it needs it and leverage parallelism



From "***Modeling Parallel Computers as Memory Hierarchies*" *by B. Alpern and L. Carter and J. Ferrante, 1993.*

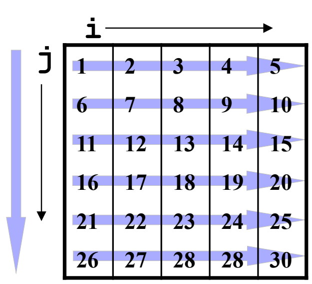


*From "***Sequoia: Programming the Memory Hierarchy*" *by Fatahalian et al., 2006.*

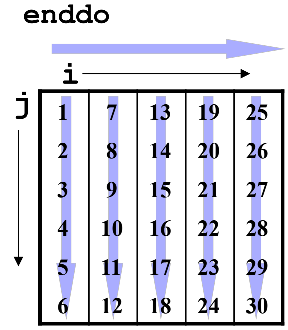
Loop Permutation for Improved Locality

Sample code: Assume Fortran's Column Major Order array layout

```
do j = 1,6                    do i = 1,5
  do i = 1,5                    do j = 1,6
     A(j,i) = A(j,i)+1            A(j,i) = A(j,i)+1
  enddo                         enddo
enddo                         enddo
```



poor cache locality



good cache locality

Loop Permutation Another Example

Idea

- Swap the order of two loops to increase parallelism, to improve spatial locality, or to enable other transformations
- Also known as **loop interchange**

Example

```
do i = 1,n
   do j = 1,n
      x = A(2,j)
   enddo
enddo
```

This access strides through a row of A

→

```
do j = 1,n
   do i = 1,n
      x = A(2,j)
   enddo
enddo
```

This code is invariant with respect to the inner loop, yielding better locality

Loop Permutation Legality

Sample code

```
do j = 1,6
  do i = 1,5
    A(j,i) = A(j,i)+1
  enddo
enddo
```

```
do i = 1,5
  do j = 1,6
    A(j,i) = A(j,i)+1
  enddo
enddo
```

Why is this legal?

- No loop-carried dependences, so we can arbitrarily change order of iteration execution
- Does the loop always have to have NO inter-iteration dependences for loop permutation to be legal?

Data Dependences

Recall

- A data dependence defines ordering relationship two between statements
- In executing statements, data dependences must be respected to preserve correctness

Example

s_1	`a := 5;`		s_1	`a := 5;`
s_2	`b := a + 1;`	$\stackrel{?}{\equiv}$	s_3	`a := 6;`
s_3	`a := 6;`		s_2	`b := a + 1;`

Dependences and Loops

Loop-independent dependences

```
do i = 1,100
   A(i) = B(i)+1
   C(i) = A(i)*2
enddo
```

Dependences within the same loop iteration

Loop-carried dependences

```
do i = 1,100
   A(i) = B(i)+1
   C(i) = A(i-1)*2
enddo
```

Dependences that cross loop iterations

Data Dependence Terminology

We say statement s_2 depends on s_1

- **True (flow) dependence**: s_1 writes memory that s_2 later reads
- **Anti-dependence**: s_1 reads memory that s_2 later writes
- **Output dependences**: s_1 writes memory that s_2 later writes
- **Input dependences**: s_1 reads memory that s_2 later reads

Notation: $s_1 \ \delta \ s_2$

- s_1 is called the **source** of the dependence
- s_2 is called the **sink** or **target**
- s_1 must be executed before s_2

Yet Another Loop Permutation Example

Consider another example

```
do i = 1,n
   do j = 1,n
      C(i,j) = C(i+1,j-1)
    enddo
enddo
```

```
do j = 1,n
   do i = 1,n
      C(i,j) = C(i+1,j-1)
    enddo
enddo
```

Before

(1,1)   C(1,1) = C(2,0)
(1,2)   C(1,2) = C(2,1)
... δ^a
(2,1)   C(2,1) = C(3,0)

After

(1,1)   C(1,1) = C(2,0)
(2,1)   C(2,1) = C(3,0)
... δ^f
(1,2)   C(1,2) = C(2,1)

Data Dependences and Loops

How do we identify dependences in loops?

```
do i = 1,5
   A(i) = A(i-1)+1
enddo
```

```
A(1) = A(0)+1
A(2) = A(1)+1
A(3) = A(2)+1
A(4) = A(3)+1
A(5) = A(4)+1
```

Simple view

– Imagine that all loops are fully unrolled
– Examine data dependences as before

Problems

– Impractical and often impossible
– Lose loop structure

Iteration Spaces

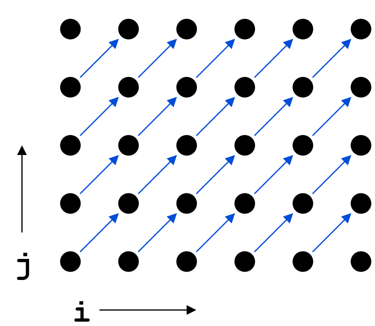
Idea

– Explicitly represent the iterations of a loop nest

Example

```
do i = 1,6
   do j = 1,5
      A(i,j) = A(i-1,j-1)+1
   enddo
enddo
```

Iteration Space



Iteration Space

– A set of tuples that represents the iterations of a loop
– Can visualize the dependences in an iteration space

Distance Vectors

Idea

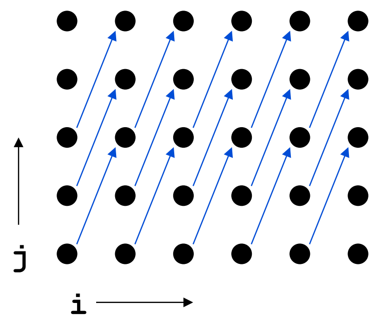
- Concisely describe dependence relationships between iterations of an iteration space
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

Definition

- $\mathbf{v} = \mathbf{i}^T - \mathbf{i}^S$

Example

```
do i = 1,6
   do j = 1,5
      A(i,j) = A(i-1,j-2)+1
   enddo
enddo
```



outer loop

Distance Vector: (1,2)

inner loop

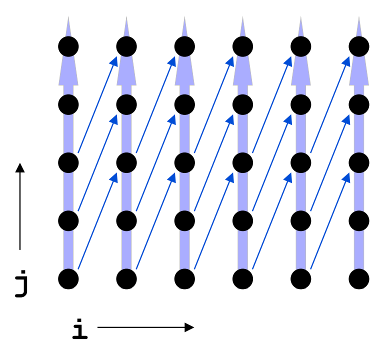
Distance Vectors and Loop Transformations

Idea

- Any transformation we perform on the loop must respect the dependences

Example

```
do i = 1,6
   do j = 1,5
      A(i,j) = A(i-1,j-2)+1
   enddo
enddo
```



Can we permute the `i` and `j` loops?

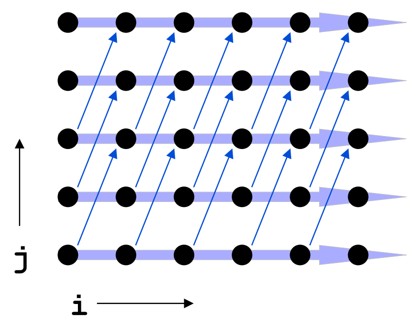
Distance Vectors and Loop Transformations

Idea

– Any transformation we perform on the loop must respect the dependences

Example

```
do j = 1,5
  do i = 1,6
     A(i,j) = A(i-1,j-2)+1
  enddo
enddo
```



Can we permute the `i` and `j` loops?

– Yes

Distance Vectors: Legality

Definition

– A dependence vector, *v*, is **lexicographically nonnegative** when the left-most entry in *v* is positive or all elements of *v* are zero
Yes: (0,0,0), (0,1), (0,2,-2)
No: (-1), (0,-2), (0,-1,1)

– A dependence vector is **legal** when it is lexicographically nonnegative (assuming that indices increase as we iterate)

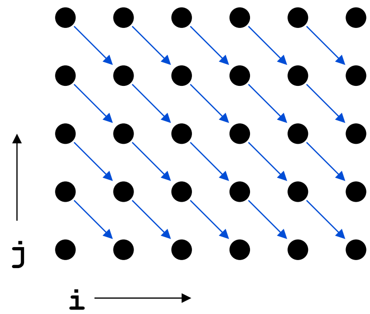
Why are lexicographically negative distance vectors illegal?

What are legal direction vectors?

Example where permutation is not legal

Sample code

```
do i = 1,6
   do j = 1,5
      A(i,j) = A(i-1,j+1)+1
   enddo
enddo
```



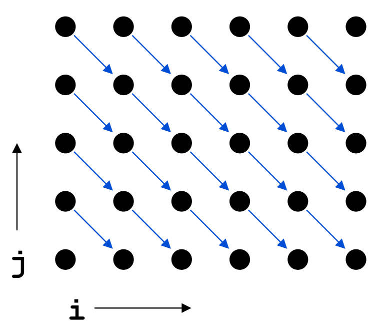
Kind of dependence: Flow

Distance vector: (1, −1)

Exercise

Sample code

```
do j = 1,5
   do i = 1,6
      A(i,j) = A(i-1,j+1)+1
   enddo
enddo
```



Kind of dependence: Anti

Distance vector: (1, -1)

Loop-Carried Dependences

Definition

– A dependence $D=(d_1,\ldots d_n)$ is **carried** at loop level i if d_i is the first nonzero element of D

Example

```
do i = 1,6
   do j = 1,6
      A(i,j) = B(i-1,j)+1
      B(i,j) = A(i,j-1)*2
   enddo
enddo
```

Distance vectors: (0,1) for accesses to **A**
(1,0) for accesses to **B**

Loop-carried dependences

– The j loop carries dependence due to **A**

– The i loop carries dependence due to **B**

Direction Vector

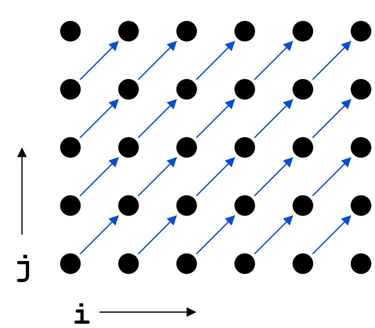
Definition

– A direction vector serves the same purpose as a distance vector when less precision is required or available

– Element i of a direction vector is <, >, or = based on whether the source of the dependence precedes, follows or is in the same iteration as the target in loop i

Example

```
do i = 1,6
   do j = 1,5
      A(i,j) = A(i-1,j-1)+1
   enddo
enddo
```



Direction vector: (<,<)

Distance vector: (1,1)

Legality of Loop Permutation

Case analysis of the direction vectors

(=,=)

The dependence is loop independent, so it is unaffected by permutation

(=,<)

The dependence is carried by the j loop.

After permutation the dependence will be (<,=), so the dependence will still be carried by the j loop, so the dependence relations do not change.

(<,=)

The dependence is carried by the i loop.

After permutation the dependence will be (=,<), so the dependence will still be carried by the i loop, so the dependence relations do not change.

Legality of Loop Interchange (cont)

Case analysis of the direction vectors (cont.)

(<,<)

The dependence distance is positive in both dimensions.

After permutation it will still be positive in both dimensions, so the dependence relations do not change.

(<,>)

The dependence is carried by the outer loop.

After interchange the dependence will be (>,<), which changes the dependences and results in an illegal direction vector, so interchange is illegal.

(>,*) (=,>)

Such direction vectors are not possible for the original loop.

Loop Interchange Example

Consider the (<,>) case

```
do i = 1,n
   do j = 1,n
      C(i,j) = C(i+1,j-1)
   enddo
enddo
```

⇒

```
do j = 1,n
   do i = 1,n
      C(i,j) = C(i+1,j-1)
   enddo
enddo
```

Before

(1,1) C(1,1) = C(2,0)

(1,2) C(1,2) = C(2,1)

. . . δ^a

(2,1) C(2,1) = C(3,0)

After

(1,1) C(1,1) = C(2,0)

(2,1) C(2,1) = C(3,0)

. . . δ^f

(1,2) C(1,2) = C(2,1)

Concepts

Touchstone apps for the class

– The Berkeley dwarf/motif categories they represent
– Operational intensity within the touchstone apps
– Data reuse within the touchstone apps
– Parallelism within the touchstone apps

Loop Transformations

– Memory layout for Fortran and C
– Loop permutation and when it is applicable
– Data dependences including distance vectors, loop carried dependences, and direction vectors

Next Time

Keep Reading

– Advanced Compiler Optimizations for Supercomputers by Padua and Wolfe

Homework

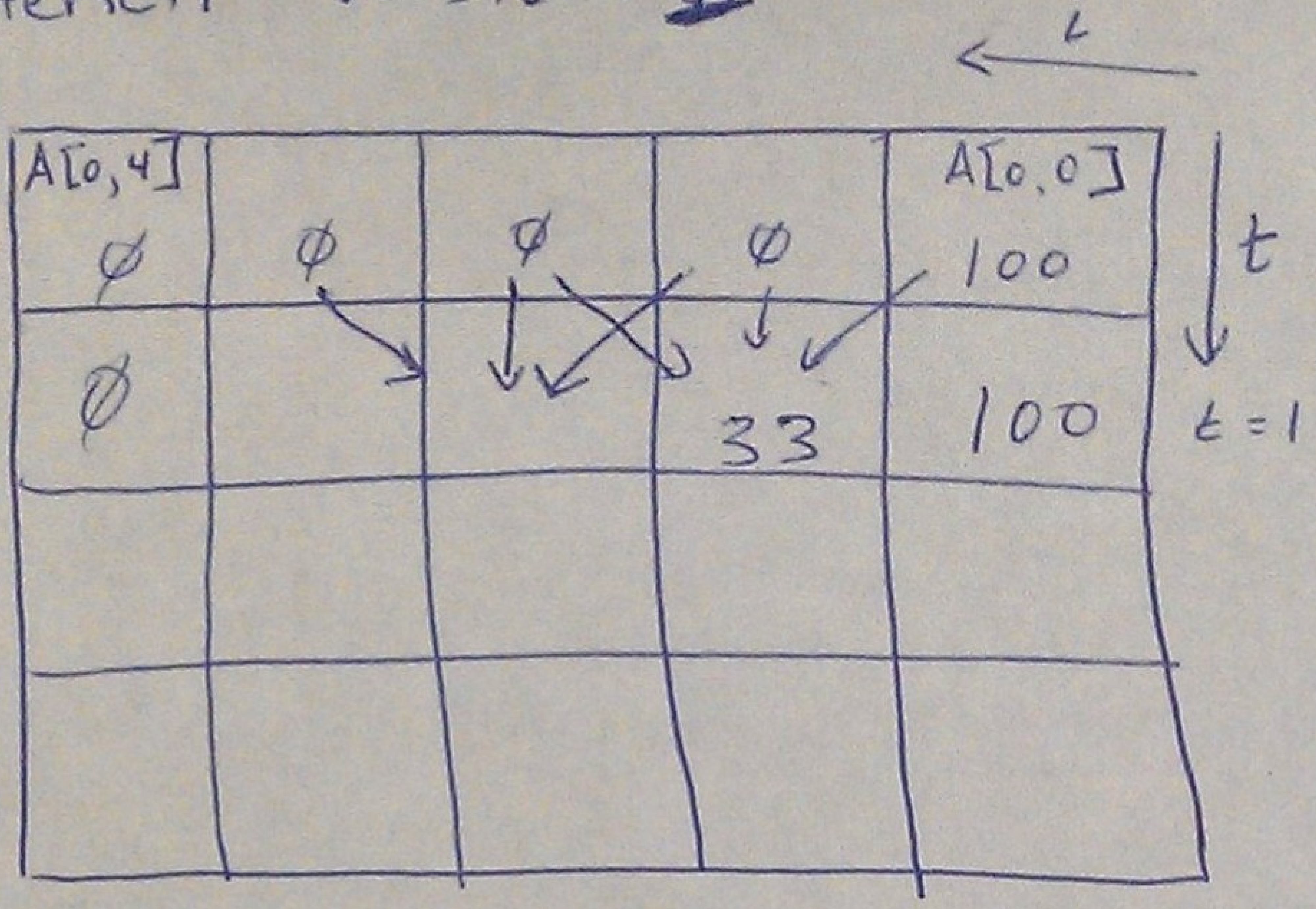
– HW0 is due Friday 1/27/12

– HW1 is due Wednesday 2/8/12

Lecture

– Parallelization and Performance Optimization of Applications

1D Stencil version 1



N=5

T=3

Operational/Arithmetic Intensity

3 flops

4 reads/writes

assume per iteration

8 bytes read

8 bytes written

$\frac{3}{16}$

memory bound

Data Reuse

every iteration reusing 2 doubles that were previously read

iteration (t, i) read A[t-1, i-1] A[t-1, i] A[t-1, i+1]

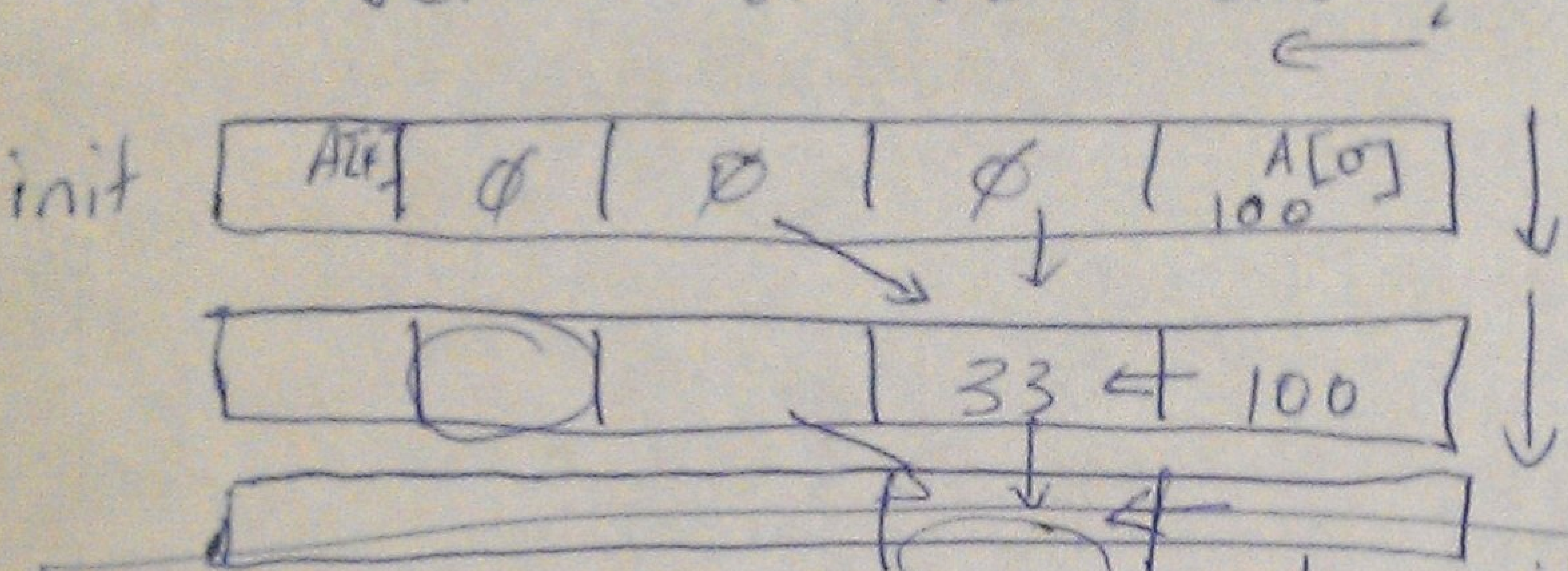
write A[t, i] reused in (t+1, i+1)

(t+1, i)

(t+1, i-1)

Parallelism

- all iteration in each row can be done in parallel version 2 1D stencil



operational/arithmetic intensity

4 flops

8 bytes assume only A[i+1] brought from memory each iteration

$4/8 = 1/2$ memory bound

data reuse

each iteration reuses A[i] & A[i-1]

parallelism

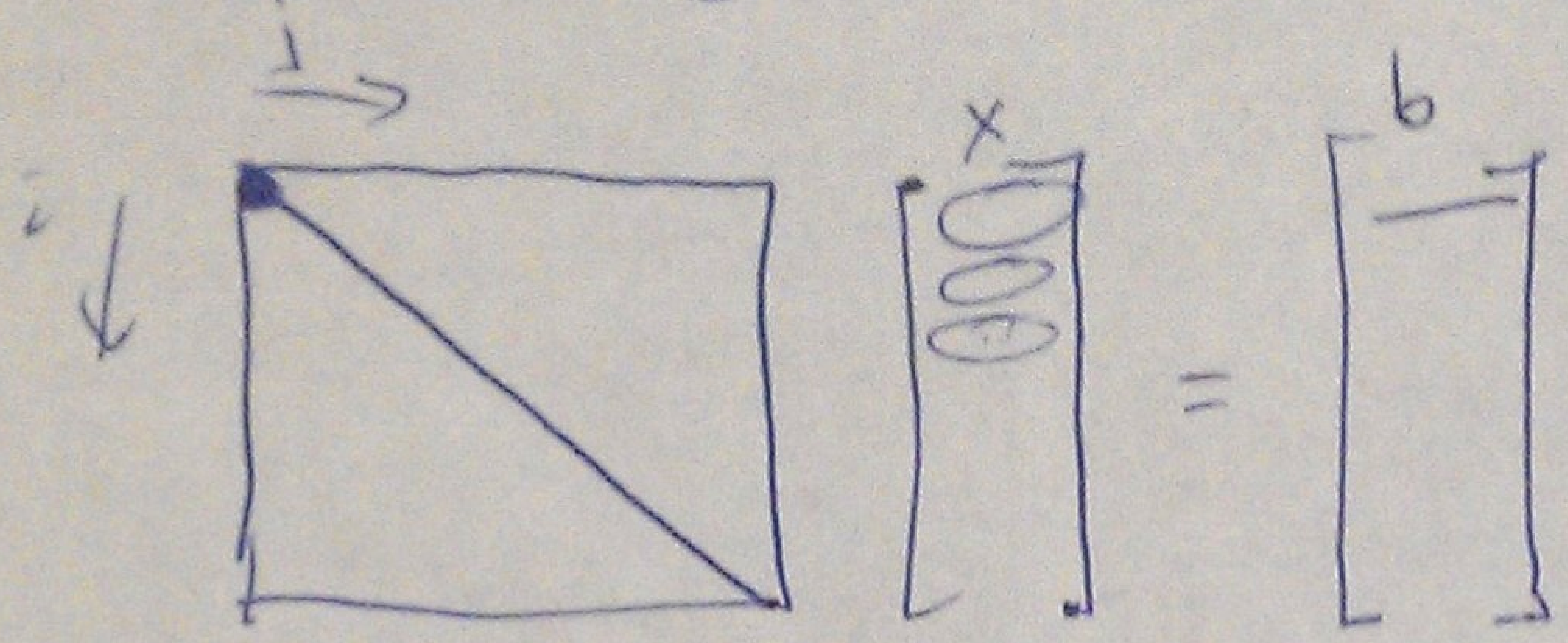
$(1,3) \parallel (2,1)$

$(1,4) \parallel (2,2)$

$\vdots$

$(t,i) \parallel (t+1, i-2)$

$$b_i = \sum_{j=1}^{N} L_{ij} x_j$$



$$x_1 = b_1 / L_{11}$$

$$x_i = b_i / \sum_{j=1}^{i-1} L_{i,j} x_j$$

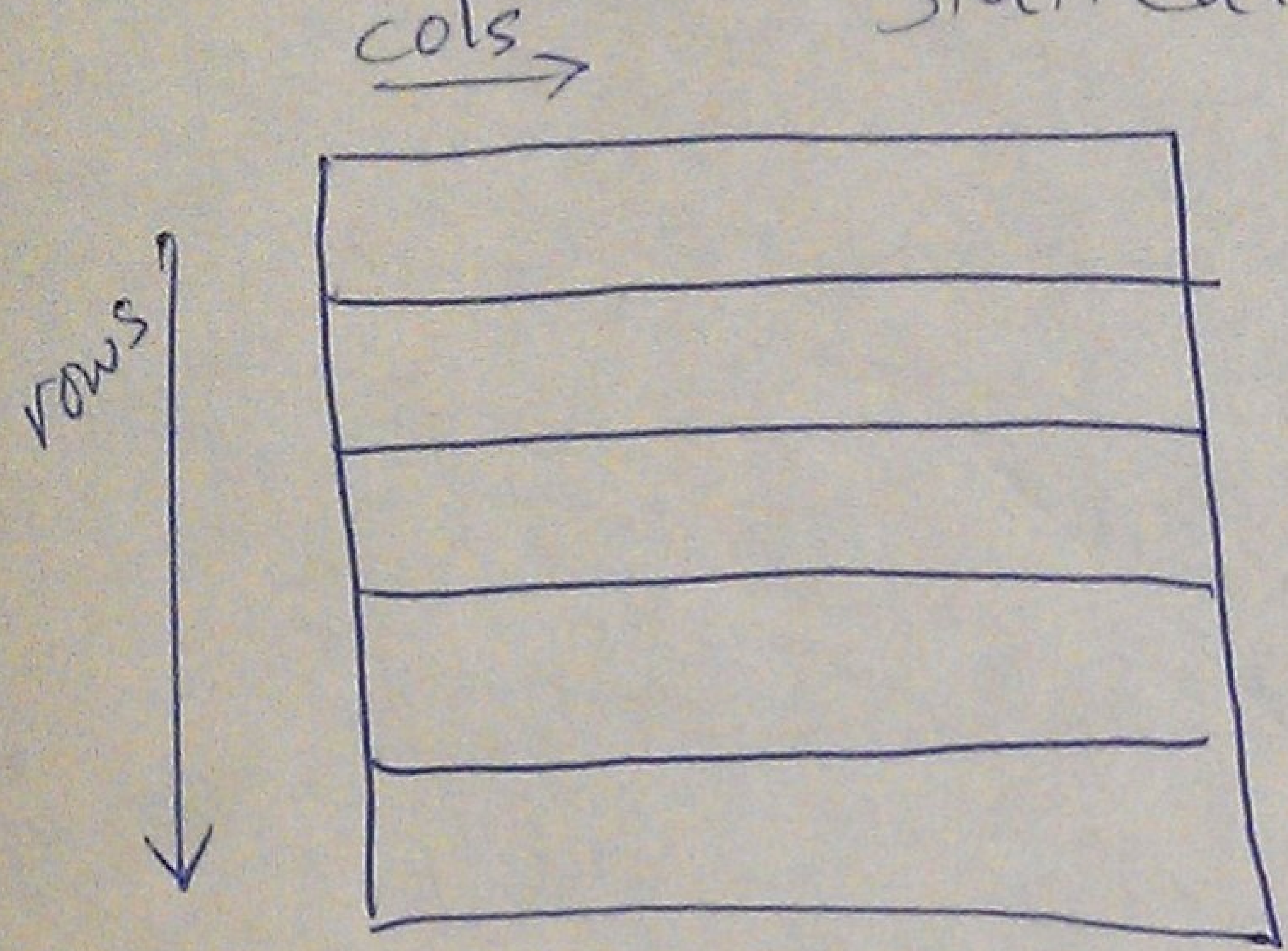
for $i = 1$ to N
    $t = 0$;
    for $j = 1$ to $i-1$
        $t \mathrel{+}= L_{ij} \underline{x_j}$
    $\underline{x_i} = b_i / t$

operational intensity
flops $= O(N^2)$
$\approx 2N^2$
memops $= O(N^2)$
$\approx 2N^2$

Data Reuse
    x_j reused in i loop

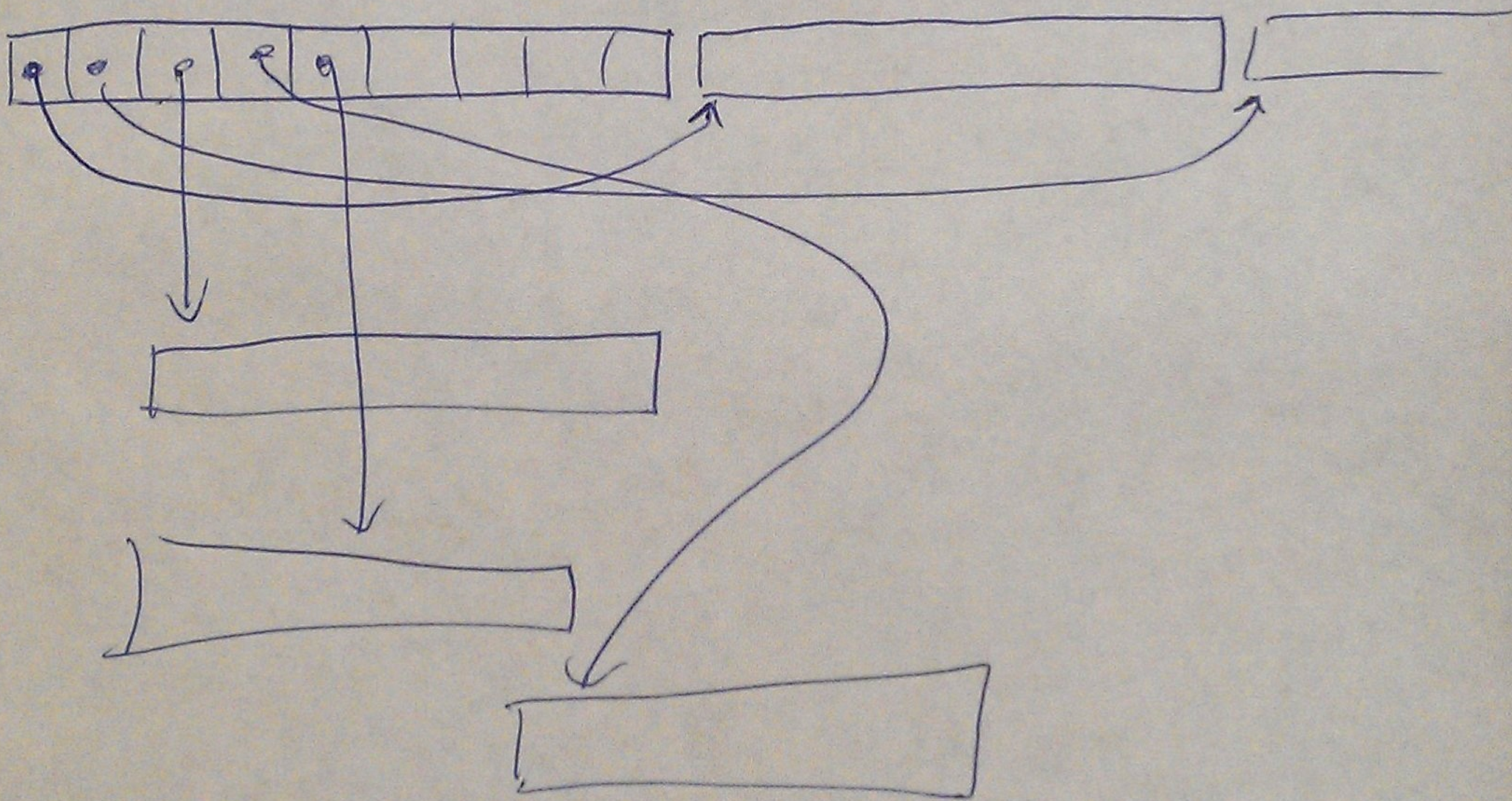
Parallelism
    i loop is serial
    reduction parallelization of inner loop

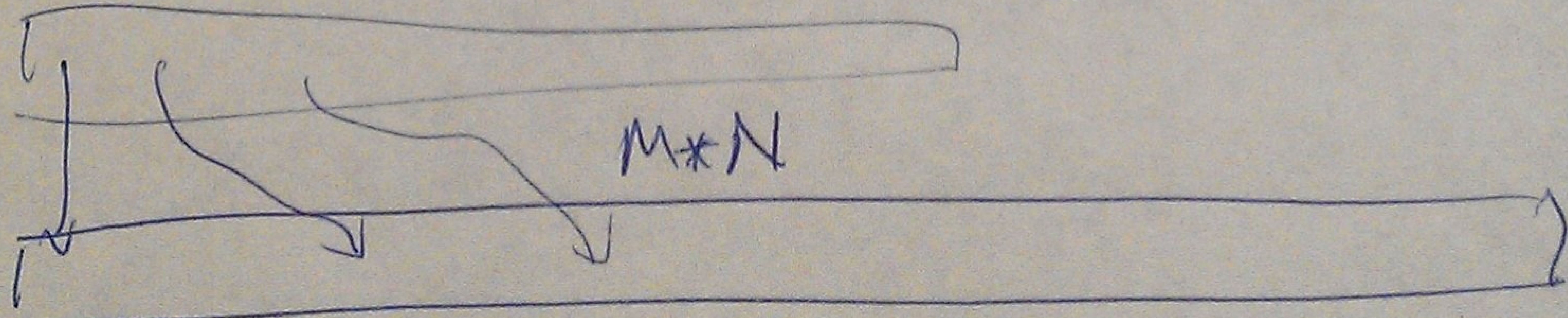
C Statically allocate



double A[N][M]

row major order





A[ ][ ]

#define A(i,j) A[i*N+j]