Class Information

- Third and last project has been posted. Due date: Monday, May 5 (last day of classes).
- Last, non-graded homework will be posted on Tuesday, April 29.
- Last lecture: Friday, May 2
- Final exam: May 8, noon to 3:00pm.
  CONFLICTS? Location will be announced later.
- Extension for homework 8 ?

A Simple Vectorizing Compiler

How to vectorize the following loops?

```
for (i=2; i<100; i++) {
  S1:  a[i] = b[i+1] + 1;
  S2:  b[i] = a[i] + 5;
}

for (i=2; i<100; i++) {
  S1:  a[i] = b[i-1] + a[i-1] + 3;
  S2:  b[i] = a[i+1] + 5;
}
```

Simple vectorizer assumptions:

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array variables
4. simple array index expressions of induction variable (i +/- c or c); can use ZIV or SIV test
5. no function calls

A Simple Vectorizing Source-to-Source Compiler

SKETCH OF BASIC ALGORITHM

Here is a basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more
2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do
   (a) if SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential
   (b) if SCC is a single statement and has no dependence cycle, distribute loop around it and generate vector code; otherwise, mark distributed loop sequential.

Loop Transformations

Goal

- modify execution order of loop iterations
- preserve data dependence constraints

Motivation

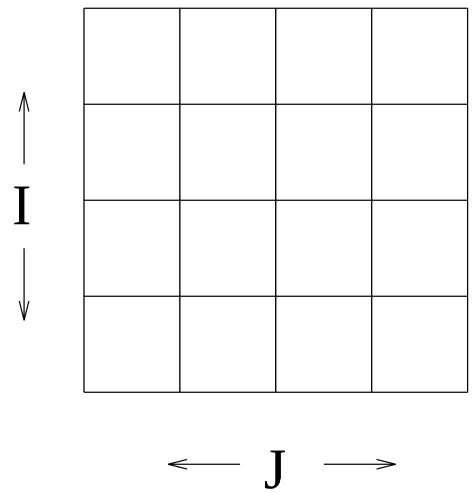
- data locality
  (increase reuse of registers, cache)
- parallelism
  (eliminate loop-carried deps, incr granularity)

Taxonomy

- loop interchange
  (change order of loops in nest)
- loop fusion
  (merge bodies of adjacent loops)
- loop distribution
  (split body of loop into adjacent loops)
- strip-mine and interchange (tiling, blocking)
  (split loop into nested loops, then interchange)

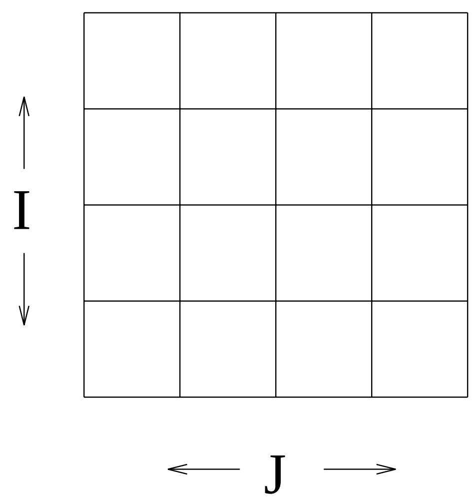
Loop Interchange

```
   do I = 1, N
     do J = 1, N
S1     A(I,J) = A(I,J-1)
S2     B(I,J) = B(I-1,J-1)
     enddo
   enddo
```



$\Longrightarrow$ loop interchange $\Longrightarrow$

```
   do J = 1, N
     do I = 1, N
S1     A(I,J) = A(I,J-1)
S2     B(I,J) = B(I-1,J-1)
     enddo
   enddo
```



Loop interchange is safe *iff*

- it does not create a lexicographically negative direction vector $(1,–1) \rightarrow (–1,1)$

$\Rightarrow$ Benefits

  - ∘ may expose parallel loops, incr granularity
  - ∘ reordering iterations may improve reuse

Loop Fusion

```
    do i = 2, N
S1    A(i) = B(i)
    do i = 2, N
S2    B(i) = A(i-1)
```

$\Longrightarrow$ loop fusion $\Longrightarrow$

```
    do i = 2, N
S1    A(i) = B(i)
S2    B(i) = A(i-1)
```

Loop fusion is safe *iff*

- no loop-independent dependence between nests is converted to a backward loop-carried dep

  (would fusion be safe if S_2 referenced `a(i+1)` ?)

$\Rightarrow$ Benefits

- reduces loop overhead
- improves reuse between loop nests
- increases granularity of parallel loop

Loop Distribution (Fission)

```
    do i = 2, N
S1    A(i) = B(i)
S2    B(i) = A(i-1)
```

$\Longrightarrow$ loop distribution $\Longrightarrow$

```
    do i = 2, N
S1    A(i) = B(i)

    do i = 2, N
S2    B(i) = A(i-1)
```

Loop distribution is safe *iff*

- statements involved in a cycle of true deps (*recurrence*) remain in the same loop, and
- if $\exists$ a dependence between two statements placed in different loops, it must be forward

$\Rightarrow$ Benefits

- necessary for vectorization
- may enable partial/full parallelization
- may enable other loop transformations
- may reduce register/cache pressure