
COMP 515: Advanced Compilation for Vector and Parallel Processors

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu



Announcements

- **Take-home midterm**
 - Will be given in class on Tuesday, Feb 24th
 - No class on Thursday, Feb 26th
 - Due by 5pm on Tuesday, March 3rd

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
 - <http://www.cs.rice.edu/~ken/comp515/>

Scalar Replacement

- Example: Scalar Replacement in case of *loop carried dependence spanning multiple iterations*

```
DO I = 1, N
  A(I) = B(I-1) + B(I+1)
ENDDO
```

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
  t3 = B(I+1)
  A(I) = t1 + t3
  t1 = t2
  t2 = t3
ENDDO
```

- One less load for each iteration for reference to B which had a loop carried input dependence spanning 2 iterations
- Invariants maintained were $t1=B(I-1); t2=B(I); t3=B(I+1)$

Eliminate Scalar Copies

```
t1 = B(0)
```

```
t2 = B(1)
```

```
DO I = 1, N
```

```
  t3 = B(I+1)
```

```
  A(I) = t1 + t3
```

```
  t1 = t2
```

```
  t2 = t3
```

```
ENDDO
```

- Unnecessary register-register copies
- Unroll loop 3 times

```
t1 = B(0)
```

```
t2 = B(1)
```

```
mN3 = MOD(N,3)
```

```
DO I = 1, mN3
```

```
  t3 = B(I+1)
```

```
  A(I) = t1 + t3
```

```
  t1 = t2
```

```
  t2 = t3
```

```
ENDDO
```

```
DO I = mN3 + 1, N, 3
```

```
  t3 = B(I+1)
```

```
  A(I) = t1 + t3
```

```
  t1 = B(I+2)
```

```
  A(I+1) = t2 + t1
```

```
  t2 = B(I+3)
```

```
  A(I+2) = t3 + t2
```

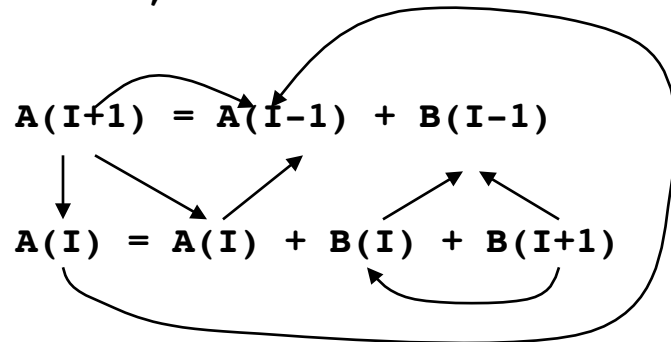
```
ENDDO
```

Preloop

Main Loop

Pruning the dependence graph

DO I = 1, N

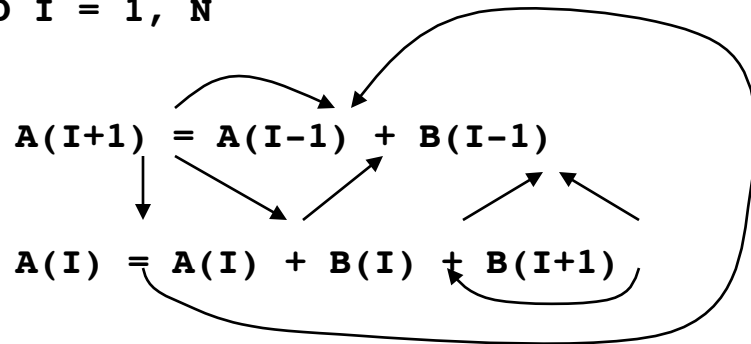


ENDDO

- Dependence pattern before pruning
- Not all edges suggest memory access savings

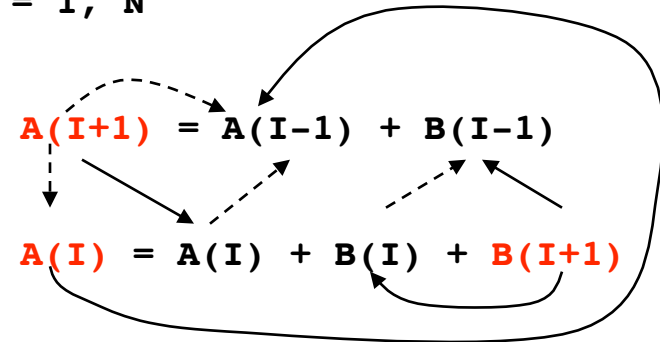
Pruning the dependence graph

DO I = 1, N



ENDDO

DO I = 1, N



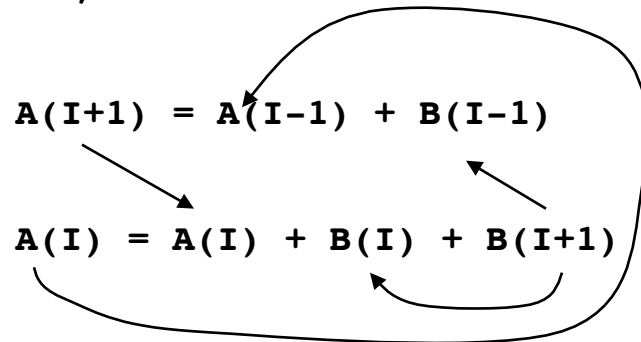
ENDDO

- Dependence pattern before pruning
- Not all edges suggest memory access savings

- Dashed edges are pruned
- Each reference has at most one predecessor in the pruned graph
- **Generator** = source of edge in pruned graph

Pruning the dependence graph

DO I = 1, N



ENDDO

- Apply scalar replacement after pruning the dependence graph

t0A = A(0); t1A0 = A(1);

tB1 = B(0); tB2 = B(1);

DO I = 1, N

t1A1 = t0A + tB1

tB3 = B(I+1)

t0A = t1A0 + tB2 + tB3

A(I) = t0A

t1A0 = t1A1

tB1 = tB2

tB2 = tB3

ENDDO

A(N+1) = t1A1

- Only one load and one store per iteration

Pruning the dependence graph

- Prune all anti dependence edges
- Prune flow and input dependence edges that do not represent a potential reuse
- Prune redundant input dependence edges
- Prune output dependence edges after rest of the pruning is done

Pruning the dependence graph

- Phase 1: Eliminate killed dependences
 - When killed dependence is a flow dependence
 - S1: $A(I+1) = \dots$
 - S2: $A(I) = \dots$
 - S3: $\dots = A(I)$
 - Store in S2 is a killing store. Flow dependence from S1 to S3 is pruned
 - When killed dependence is an input dependence
 - S1: $\dots = A(I+1)$
 - S2: $A(I) = \dots$
 - S3: $\dots = A(I-1)$
 - Store in S2 is a killing store. Input dependence from S1 to S3 is pruned

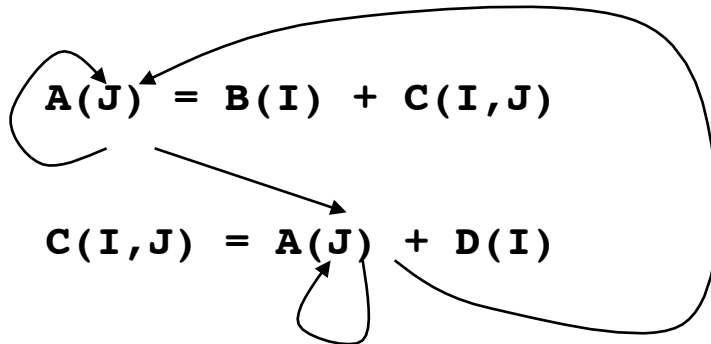
Pruning the dependence graph

- Phase 3: Find name partitions and eliminate input dependences
 - Use Typed Fusion
 - References as vertices
 - An edge joins two references
 - Output and anti- dependences are bad edges
 - Name of array as type
- Eliminate input dependences between two elements of same name partition unless source is a generator

Pruning the dependence graph

- **Special cases**
 - Reference is in a dependence cycle in the loop

DO I = 1, N



ENDDO

- Assign single scalar to the reference in the cycle
- Replace $A(J)$ by a scalar tA and insert $A(J)=tA$ before or after the loop depending on upward/downward exposed occurrence

Pruning the dependence graph

- **Special cases: Inconsistent dependences**

```
DO I = 1, N
  A(I) = A(I-1) + B(I)
  A(J) = A(J) + A(I)
ENDDO
```

- **Store to A(J) kills A(I)**

- **Only one scalar replacement possible**

```
DO I = 1, N
  tAI = A(I-1) + B(I)
  A(I) = tAI
  A(J) = A(J) + tAI
ENDDO
```

- **This code can be improved substantially by index set splitting**

Pruning the dependence graph

```
DO I = 1, N
  tAI = A(I-1) + B(I)
  A(I) = tAI
  A(J) = A(J) + tAI
ENDDO
```

- Split this loop into three separate parts
 - A loop up to J
 - Iteration J
 - A loop after iteration J to N

```
tAI = A(0); tAJ = A(J)
JU = MAX(J-1, 0)
```

```
DO I = 1, JU
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
ENDDO
```

```
IF (J.GT.0.AND.J.LE.N) THEN
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
  tAI = tAJ
ENDIF
```

```
DO I = JU+2, N
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
ENDDO
A(J) = tAJ
```

Scalar Replacement: Putting it together

1. Prune dependence graph; Apply typed fusion
2. Select a set of name partitions using register pressure moderation
3. For each selected partition
 - A) If non-cyclic, replace using set of temporaries
 - B) If cyclic replace reference with single temporary
 - C) For each inconsistent dependence
 - Use index set splitting or insert loads and stores
4. Unroll loop to eliminate scalar copies

Scalar Replacement: Case A

```
DO I = 1, N
```

```
A(I+1) = A(I-1) + B(I-1)
```

```
A(I) = A(I) + B(I) + B(I+1)
```

```
ENDDO
```



```
t0A = A(0); t1A0 = A(1);
```

```
tB1 = B(0); tB2 = B(1)
```

```
DO I = 1, N
```

```
t1A1 = t0A + tB1
```

```
tB3 = B(I+1)
```

```
t0A = t1A0 + tB3 + tB2
```

```
A(I) = t0A
```

```
t1A0 = t1A1
```

```
tB1 = tB2
```

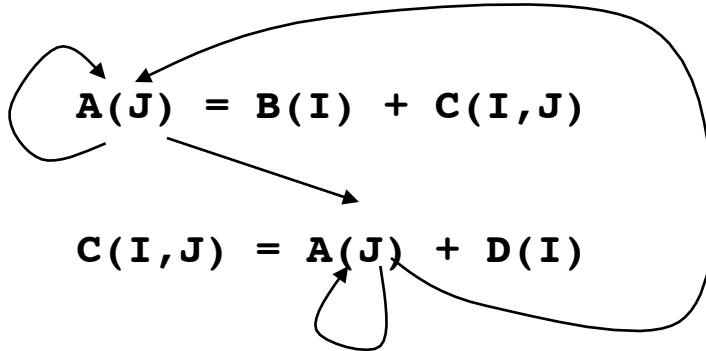
```
tB2 = tB3
```

```
ENDDO
```

```
A(N+1) = t1A1
```

Scalar Replacement: Case B

```
DO I = 1, N
```



```
ENDDO
```

- replace with single temporary...

```
DO I = 1, N
```

```
    tA = B(I) + C(I, J)
```

```
    C(I, J) = tA + D(I)
```

```
ENDDO
```

```
A(J) = tA
```

-

Scalar Replacement: Case C

```
DO I = 1, N
  tAI = A(I-1) + B(I)
  A(I) = tAI
  A(J) = A(J) + tAI
ENDDO
```

- Split this loop into three separate parts
 - A loop up to J
 - Iteration J
 - A loop after iteration J to N

```
tAI = A(0); tAJ = A(J)
JU = MAX(J-1, 0)
```

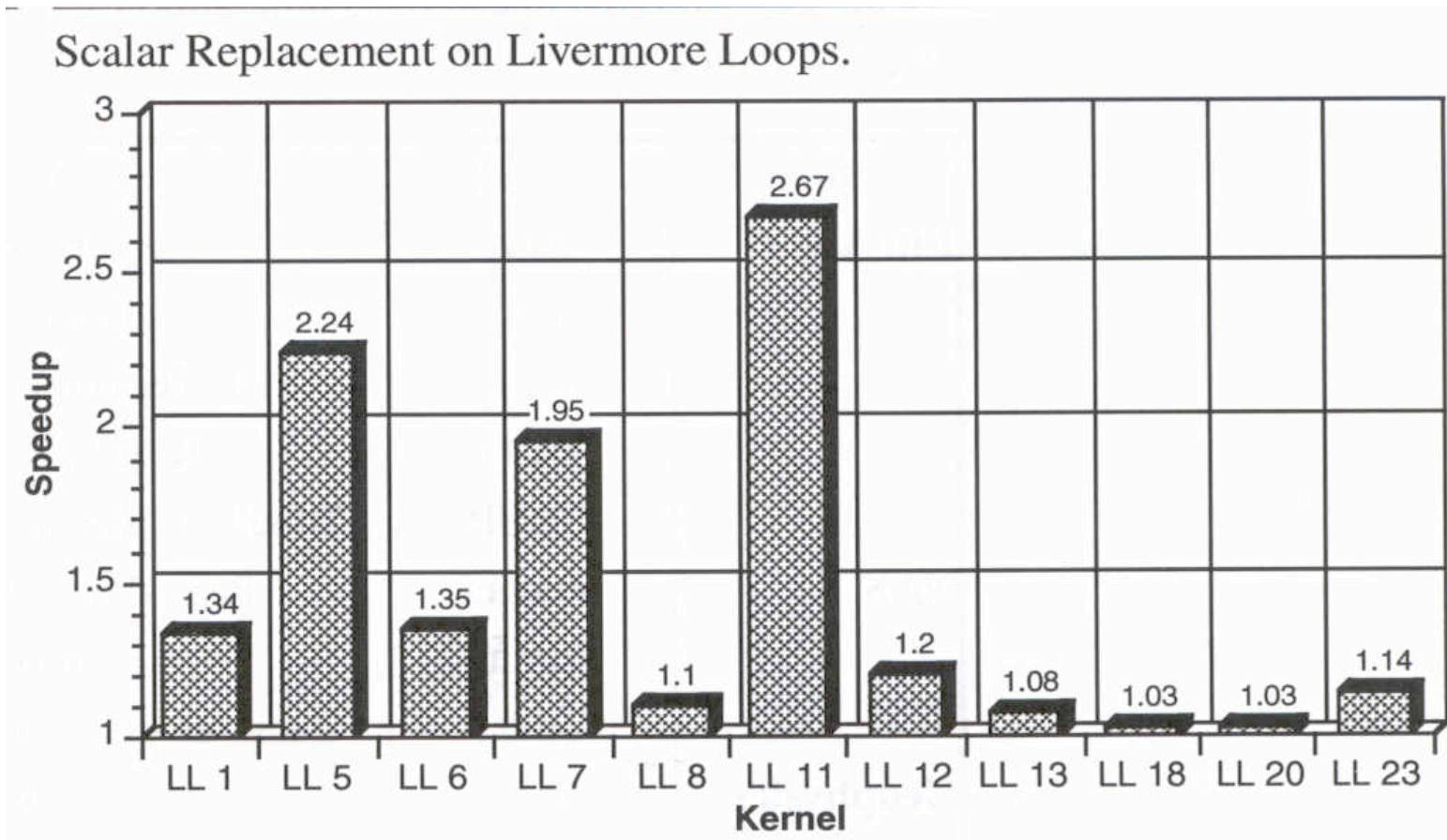
```
DO I = 1, JU
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
ENDDO
```

```
IF (J.GT.0.AND.J.LE.N) THEN
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
  tAI = tAJ
ENDIF
```

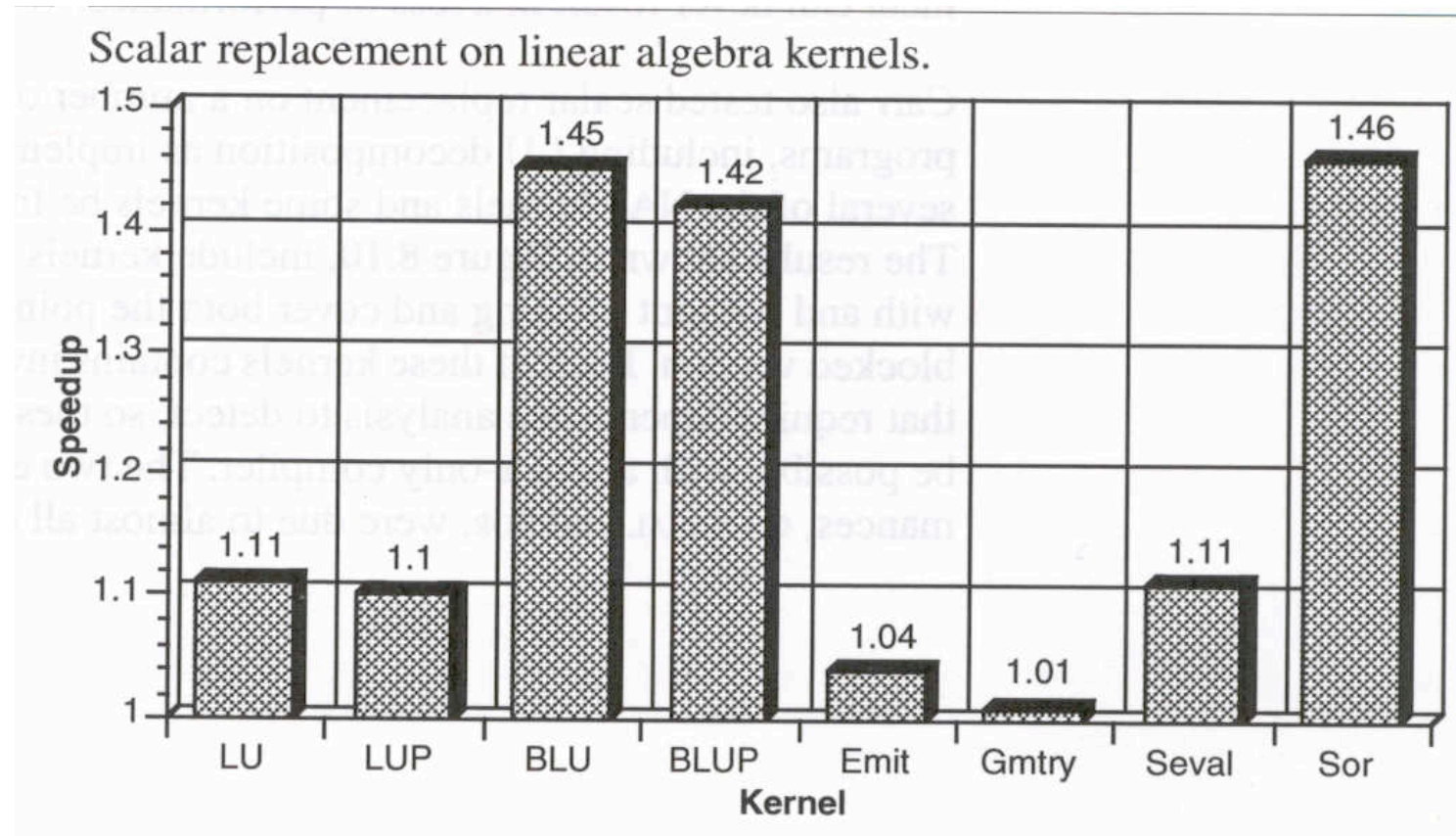
```
DO I = JU+2, N
  tAI = tAI + B(I); A(I) = tAI
  tAJ = tAJ + tAI
ENDDO
```

```
A(J) = tAJ
```

Experiments on Scalar Replacement



Experiments on Scalar Replacement



Unroll-and-Jam

```
DO I = 1, N*2
  DO J = 1, M
    A(I) = A(I) + B(J)
  ENDDO
ENDDO
```

- Can we achieve reuse of references to B ?
- Use transformation called Unroll-and-Jam

```
DO I = 1, N*2, 2
  DO J = 1, M
    A(I) = A(I) + B(J)
    A(I+1) = A(I+1) + B(J)
  ENDDO
ENDDO
```

- Unroll outer loop twice and then fuse the copies of the inner loop
- Brought two uses of B(J) together

Unroll-and-Jam

```
DO I = 1, N*2, 2
  DO J = 1, M
    A(I) = A(I) + B(J)
    A(I+1) = A(I+1) + B(J)
  ENDDO
ENDDO
```

- Apply scalar replacement on this code

```
DO I = 1, N*2, 2
  s0 = A(I)
  s1 = A(I+1)
  DO J = 1, M
    t = B(J)
    s0 = s0 + t
    s1 = s1 + t
  ENDDO
  A(I) = s0
  A(I+1) = s1
ENDDO
```

- Half the number of loads as the original program

Legality of Unroll-and-Jam

- Is unroll-and-jam always legal?

```
DO I = 1, N*2
  DO J = 1, M
    A(I+1,J-1) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

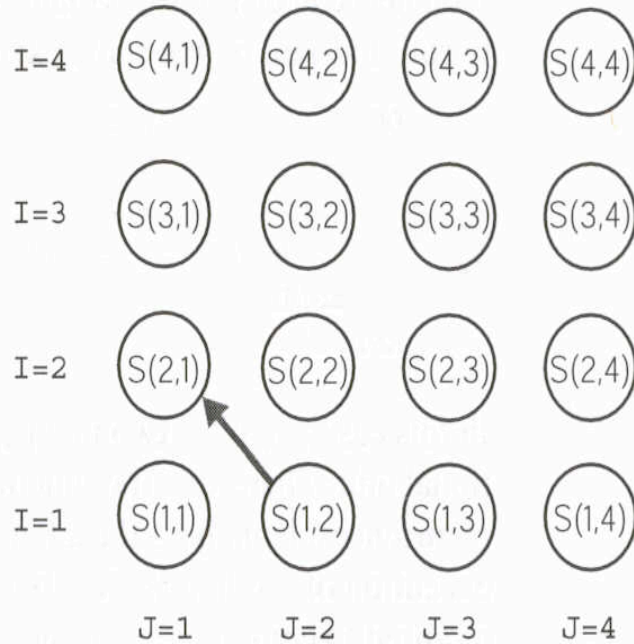
```
DO I = 1, N*2, 2
  DO J = 1, M
    A(I+1,J-1) = A(I,J) + B(I,J)
    A(I+2,J-1) = A(I+1,J) + B(
I+1,J)
  ENDDO
ENDDO
```

- Apply unroll-and-jam

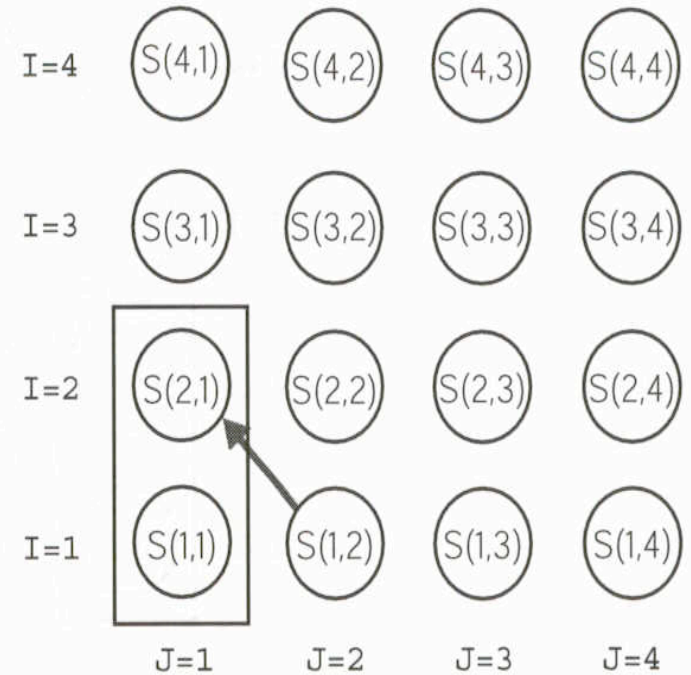
- This is wrong!!!

Legality of Unroll-and-Jam

Legality of unroll-and-jam



Legality of unroll-and-jam.



Legality of Unroll-and-Jam

- Direction vector in this example was (\langle, \rangle)
 - This makes loop interchange illegal
 - Unroll-and-Jam is loop interchange followed by unrolling inner loop followed by another loop interchange
- But does loop interchange illegal imply unroll-and-jam illegal ?
NO

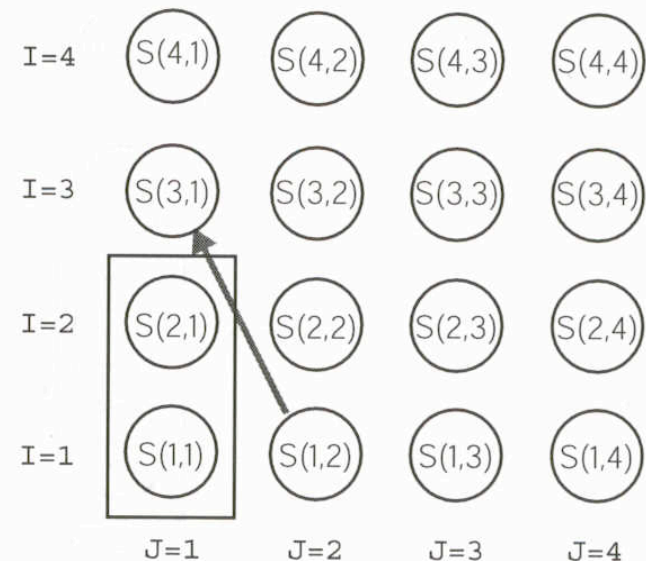
Legality of Unroll-and-Jam

- Consider this example

```
DO I = 1, N*2
  DO J = 1, M
    A(I+2,J-1) = A(I,J) +
      B(I,J)
  ENDDO
ENDDO
```

- Direction vector is (\langle, \rangle) ; still unroll-and-jam possible

Legality of unroll-and-jam.



Conditions for legality of unroll-and-jam

- **Definition:** Unroll-and-jam to factor n consists of unrolling the outer loop $n-1$ times and fusing those copies together.
- **Theorem:** An unroll-and-jam to a factor of n is legal iff there exists no dependence with direction vector (\langle, \rangle) such that the distance for the outer loop is less than n .

Unroll-and-jam Algorithm

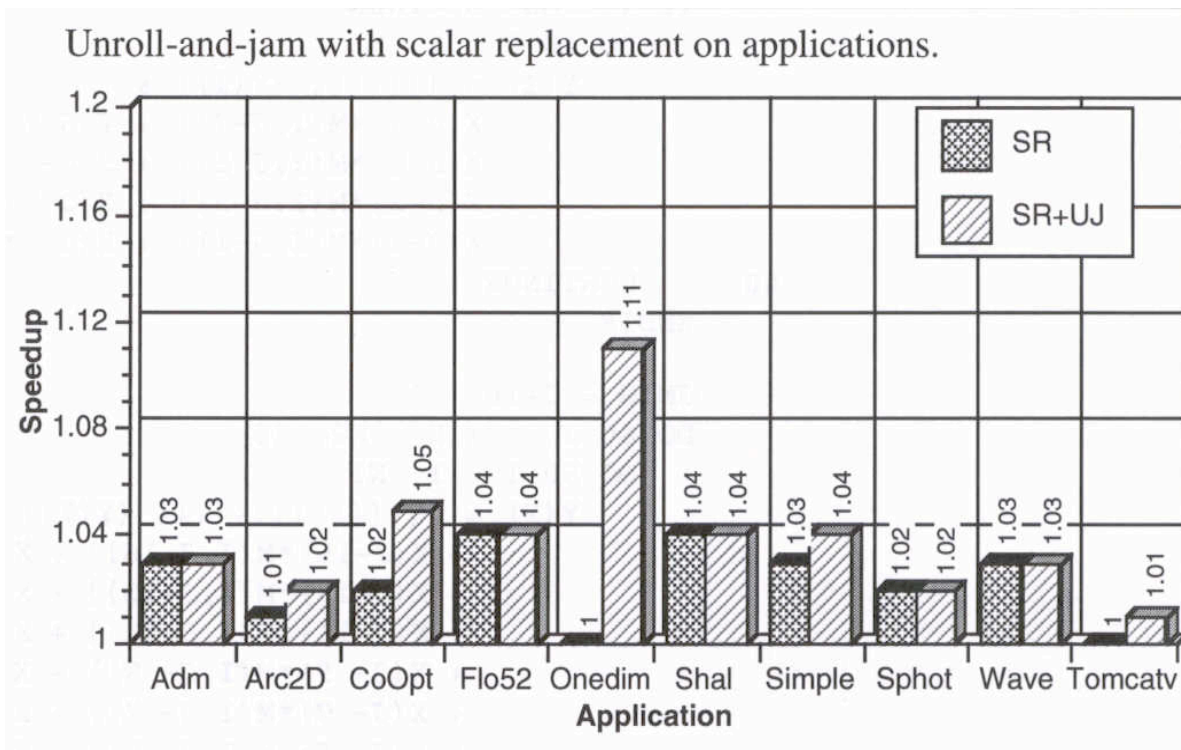
1. Create preloop
2. Unroll main loop m (the unroll-and-jam factor) times
3. Apply typed fusion to loops within the body of the unrolled loop
4. Apply unroll-and-jam recursively to the inner nested loop

Unroll-and-jam example

```
DO I = 1, N
  DO K = 1, N
    A(I) = A(I) + X(I,K)
  ENDDO
  DO J = 1, M
    DO K = 1, N
      B(J,K) = B(J,K) + A(I)
    ENDDO
  ENDDO
  DO J = 1, M
    C(J,I) = B(J,N)/A(I)
  ENDDO
ENDDO
```

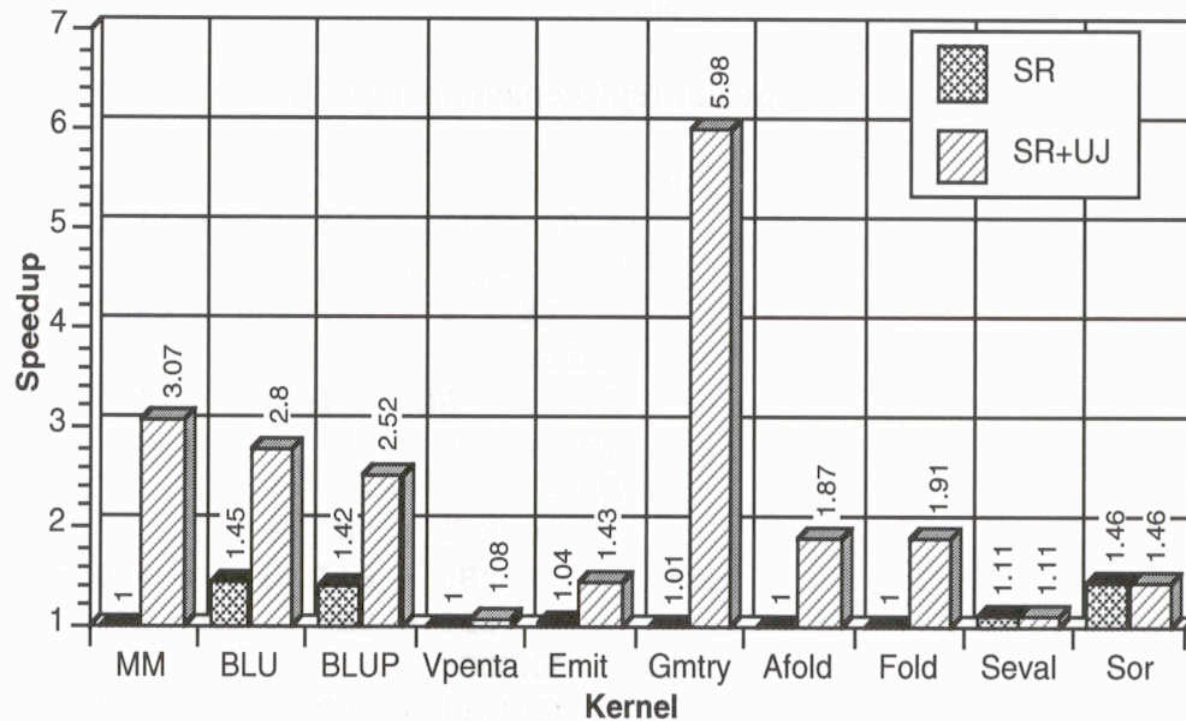
```
DO I = mN2+1, N, 2
  DO K = 1, N
    A(I) = A(I) + X(I,K)
    A(I+1) = A(I+1) + X(I+1,K)
  ENDDO
  DO J = 1, M
    DO K = 1, N
      B(J,K) = B(J,K) + A(I)
      B(J,K) = B(J,K) + A(I+1)
    ENDDO
    C(J,I) = B(J,N)/A(I)
    C(J,I+1) = B(J,N)/A(I+1)
  ENDDO
ENDDO
```

Unroll-and-jam: Experiments



Unroll-and-jam: Experiments

Unroll and jam with scalar replacement on kernels.



Conclusion

- We have learned two memory hierarchy transformations:
 - scalar replacement
 - unroll-and-jam
- They reduce the number of memory accesses by maximum use of processor registers

Homework

- Reading list for next class
 - Review Chapters 2 - 7 for Midterm Exam
- Homework assignment for discussion in next class
 - Exercise 8.2