CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

DEPENDENCE TRANSFORMS

The slides adapted from Vikram Adve

Motivation

Memory hierarchy optimizations
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

Tiling
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synchronization overhead

Software Prefetching
Goal: Prefetch predictable accesses k iterations ahead

Software Pipelining
Goal: Extract ILP from multiple consecutive iterations

Automatic parallelization Also, auto-vectorization
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code

Reordering Transformation

Definition. Legal Transformation preserves the meaning of that program, i.e., **all externally visible outputs are identical to the original program**, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

Theorem. A ***reordering*** transformation that preserves all data dependences in a program is a ***legal*** transformation.

For discussion, see Allen and Kennedy book.

Dependence Distance

Dependence Distance: If there is a dependence from statement S1 on iteration I and statement S2 on iteration I' then the corresponding dependence distance vector is

$$d_{I,I'} = [I'_1 - I_1, \ldots I'_k - I_k]$$

Note: Computing distance vectors is harder than testing dependence

Dependence Distance

Direction Vector: For a distance vector of the form $d_{I,I'} = [I'_1 - I_1, \dots, I'_k - I_k]$ the corresponding direction vector is $\delta_{I,I'} = [\delta_1, \dots, \delta_k, \dots \delta_m]$, where

$$\delta_k = \begin{cases} - , & if\ I'_k - I_k < 0 \\ +, & if I'_k - I_k > 0 \\ =, & if I'_k - I_k = 0 \\ * , & if \text{ sign } +, - , = \end{cases}$$

Note: **I** < **J** iff the leftmost non-'=' entry in δ(**I**, **J**) is '+'.

- We use the property of lexicographical ordering

Loop-Carried Dependence

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **I'** and **d(I,I')>0** .

```
do i = 1 to N
    A(i+1) = B(i)
    B(i+1) = A(i)
enddo
```

Level of loop-carried dependence is the leftmost non-“=“ sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body

Reordering Transformations

Name	Purpose	Benefit
	Preprocessing transformations	
Loop normalization	Make loops canonical	Simplify,improve dep. analysis
Ind. var. substitution	Identify aux. induction vars	Improve dependence information
Scalar expansion	Replace scalar with array	Eliminate spurious dependences
Scalar/array privatization	Treat var. as iteration-private	Eliminate spurious dependences
Variable renaming	Use multiple copies of vars	Eliminate anti- and output-dependences
Reduction recognition	Recognize reductions	Ignore special-case dependences
	Reordering transformations	
Loop interchange	Change loop nesting order	Cache, parallelism, vectorization
Loop strip-mining	Make 2 nested loops	”
Loop skewing	Change wavefront loop to parallel	Improve loop parallelism
Loop reversal	Run loop backwards	Reduce array storage
Index set splitting	Break loop by index space	Remove some deps.
Loop distribution	Break loop by statements	Simplify parallelization, vectorization
Loop alignment	Change carried to indep.	Simplify parallelization, vectorization
Loop fusion	Join loops by statements	Improve cache reuse

Math Intermezzo: Unimodular Matrix

A matrix T is unimodular iff it is a square integer matrix with determinant +1 or -1

$$\begin{bmatrix}\mathbf{0} & \mathbf{1}\\ \mathbf{1} & \mathbf{0}\end{bmatrix} \quad \begin{bmatrix}\mathbf{1} & \mathbf{0}\\ \mathbf{1} & \mathbf{1}\end{bmatrix}$$

These properties will help us compose transformations:

- Product of two unimodular matrices is also unimodular
- Its inverse is also unimodular

For each integer vector x, a unimodular matrix T maps it into a **unique vector** y = Tx

Loop Transformations and Matrices

A transformation is called ***unimodular*** if the matrix T is unimodular (i.e., square integer matrix with determinant +1 or -1)

Loop interchange: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \vec{t} = \vec{0}$

Loop reversal: $T = [-1], \vec{t} = (U_1 - 1)$

Legality of the transformation: $T \cdot \vec{\imath} \geq 0$

Examples of Unimodular Transformations

Interchange

```
for i=2 to N
   for j=2 to M-1
       A[i,j] = A[i,j]*2
    end for
end for
```

```
for j=2 to M-1
   for i=2 to N
       A[i,j] = A[i,j]*2
    end for
end for
```

Transform matrix

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

Reversal

```
for k=1 to L
  A[i,j,k] = A[i,j-1,k+1]
         + A[i-1,j,k+1]
endfor
```

```
for k=L to 1 step -1
  A[i,j,k] = A[i,j-1,k+1]
         + A[i-1,j,k+1]
endfor
```

$$[k'] = [-1][k] + L$$

Skew

```
for i=2 to N
  for j=2 to N
    A[i,j] = A[i-1,j]
           + A[i,j-1]
  end for
end for
```

```
for i=2 to N
  for jj=i+2 to i+N
    A[i,jj-i] = A[i-1,jj-i]
              + A[i,jj-i-1]
  end for
end for
```

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

Legality of Unimodular Transformations

A transformed loop nest is equivalent to the original if it preserves all dependencies. A transformation between these two nets is legal if the nests are equivalent.

Let D be the set of distance vectors of a loop nest. A unimodular transformation T is legal if and only if

$$\forall \boldsymbol{d} \in \boldsymbol{D} \ \ . \ \ \boldsymbol{T} \cdot \boldsymbol{d} \geq \mathbf{0}$$

Proof sketch (from Banerjee, Unimodular Transformations 2011):

Consider loop body S of the original nest and S’ of the transformed one. Two iterations *S(I)* and *S(I’)* in the original nest become *S’(TI)* and *S’(TI’)* in the transformed. *S’(TI)* precedes *S’(TI’)* iff $T \cdot I' \geq T \cdot I$.

“if part”: For each d, assume $T \cdot d \geq 0$. Consider that a statement *S(I’)* in iteration *I’* depend on the statement *S(I)*. Because $d = I' - I$ is the distance vector in the original loop, we get $T \cdot I' - T \cdot I = T(I' - I) \geq 0$. With this we get that all dependencies are preserved in the transformed loop., i.e. the two loop nests are equivalent.

“only-if part”: Assume the transformation is legal. Let *d=I’-I* denote a distance in the original loop (and the statement in the iteration *’I* depends on the one in iteration *I*. By hypothesis (the loop nests are equivalent), $T \cdot I' \geq T \cdot I$, so then $T \cdot I' - T \cdot I \geq 0$ and so $T \cdot (I' - I) = T \cdot d \geq 0$

Loop Interchange

Informal Definition: Change nesting order of loops in a **perfect loop nest**, with no other changes.

```
for i=2 to N
   for j=2 to M-1
       A[i,j] = A[i,j]*2
    end for
end for
```

```
for j=2 to M-1
   for i=2 to N
       A[i,j] = A[i,j]*2
    end for
end for
```

Uses of Loop Interchange

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)

Loop Interchange

Direction Vectors and Loop Interchange:

If δ is a direction vector of a particular dependence S1 → S2 in a loop nest and the order of loops in the loop nest is permuted, then the same permutation can be applied to δ to obtain the new direction vector for the conflicting instances of S1 and S2

Direction Matrix: A matrix where each row is the direction vector of a single dependence, i.e.,

each row ↔ a dependence

each column ↔ a loop

Loop Interchange Properties

Legality: A permutation of the loops in a perfect nest is legal iff the direction matrix, <u>after the permutation is applied</u>, has no “**-**” direction as the leftmost non-“=“ direction in any row

- Recall, for legality the vector after transformation should be lexicographically greater than “0” vector.
- Some more intuition: To preserve the dependencies, consider the cases before transformation of (=,=) [independent], (=,+) and (+,=) [the dependence is still carried but by the outer (resp. inner loops)], (+,+) [Dependence is still carried]. But (+, -) is illegal since the antidependence turns into a true dependence

Profitability: machine-dependent:

1. vector machines
2. parallel machines
3. caches with single outstanding loads
4. caches with multiple outstanding loads

Direction Matrix

Direction Matrix:

each row ↔ a dependence

each column ↔ a loop

Sp→Sq: A[i,j]/A[i,j]	=	=
Sp→Sq: A[i,j]/A[i-1,j]	+	=
Sq→Sp: B[i,j]/B[i-1,j-1]	+	+

```
for i = 2 to N
   for j = 2 to M-1
Sp:     A[i,j] = B[i-1,j-1]
Sq:     B[i,j] = A[i,j] + A[i-1,j]
   endfor
endfor
```

Direction Matrix (Illegal)

Direction Matrix:

each row ↔ a dependence

each column ↔ a loop

Sp→Sq: A[i,j]/A[i,j]	=	=
Sp→Sq: A[i,j]/A[i-1,j+1]	+	-
Sq→Sp: B[i,j]/B[i-1,j-1]	+	+

✖

```
for i = 2 to N
   for j = 2 to M-1
Sp:     A[i,j] = B[i-1,j-1]
Sq:     B[i,j] = A[i,j] + A[i-1,j+1]
   endfor
endfor
```

Applying Loop Interchange

1. **Single** ’+’ entry: a “serial loop”
   - Move loop outermost for vectorization
   - Move loop innermost for parallelization

2. **Multiple** ’+’ entries: Outermost one carries dependence
   - Loop carrying the dependence *changes* after permutation!
   - May still benefit by moving carried-dependences to the outermost loop

Example

```
for i = 1 to n
  for j = 1 to m
     A[i+1, j] = A[i, j] + B[i, j]
  end for
end for
```

```
for i = 1 to n
  for j = 1 to m // vectorize
     A[i+1, j] = A[i, j]
               + B[i, j]
  end for
end for
```

```
parallel for j = 1 to m
   for i = 1 to n
       A[i+1, j] = A[i, j]
                 + B[i, j]
  end for
end for
```

Loop Reversal

Informal Definition: Reverse the order of execution of the iterations of a loop

```
for i=2 to N
  for j=2 to M-1
    for k=1 to L
      A[i,j,k] = A[i,j-1,k+1]
             + A[i-1,j,k+1]
    endfor
  endfor
endfor
```

```
for i=2 to N
  for j=2 to M-1
    for k=L to 1 step -1
      A[i,j,k] = A[i,j-1,k+1]
             + A[i-1,j,k+1]
    endfor
  endfor
endfor
```

Legality of Loop Reversal

The loop that is reversed ***should not carry dependence***

Recall, ***Legality***: the vector after transformation should be lexicographically greater than "0" vector.

E.g., $(1, -1) \succ (0,0)$ but $(-1, 1) \prec (0,0)$

In our case, two dependencies:

$$(1) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} = \\ + \\ - \end{bmatrix} = \begin{bmatrix} \textcircled{=} \\ + \\ + \end{bmatrix} \succ 0 \qquad (2) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} + \\ = \\ - \end{bmatrix} = \begin{bmatrix} \textcircled{+} \\ = \\ + \end{bmatrix} \succ 0$$

Uses of Loop Reversal

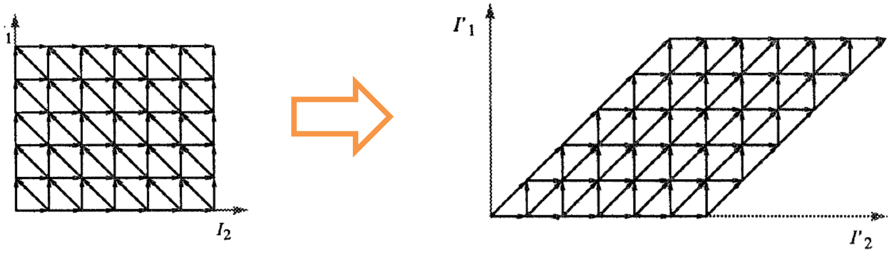
Convert a ’-’ to a ’+’ in a direction vector to enable other transformations, e.g., loop interchange.

Scalarize a vector statement (e.g., in Fortran 90) by ensuring that values are read before being written.

- Vectorized code: `A[2:64] = A[1:63] * e`
- Scalarized code:

```
for i = 64 to 2 step -1
    A[i] = A[i-1] * e
endfor
```

Loop Skewing



Informal Definition: Increase dependence distance by n by substituting loop index j with jj = j + n ∗ i.

E.g., For n = 1, we use jj = j + 1

```
for i=2 to N
  for j=2 to N
    A[i,j] = A[i-1,j]
           + A[i,j-1]
  end for
end for
```

```
for i=2 to N
  for jj=i+2 to i+N
    A[i,jj-i] = A[i-1,jj-i]
              + A[i,jj-i-1]
  end for
end for
```

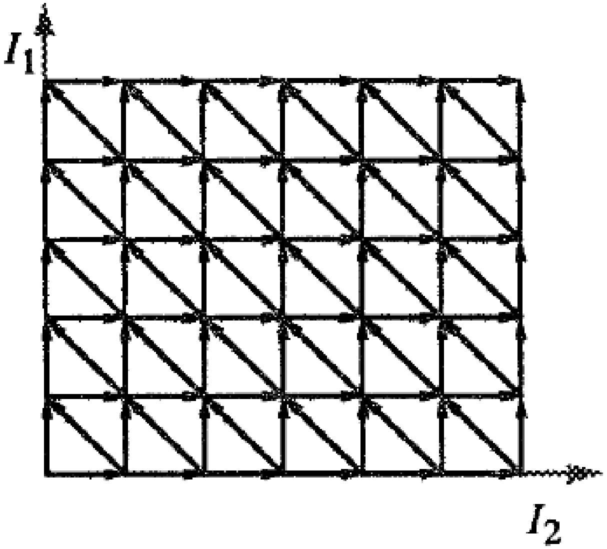
- Improve parallelism by converting ’=’ to ’+’ in a direction vector
- Improve vectorization in a similar way
- (Rarely) Could be used to *simplify* index expressions

Skewing: Full Example

from A Data Locality Optimizing Algorithm, Wolf & Lam 1991.

```
for $I_1$ := 0 to 5 do
  for $I_2$ := 0 to 6 do
    A[$I_2+1$] := 1/3 * (A[$I_2$] + A[$I_2+1$] + A[$I_2+2$]);
```

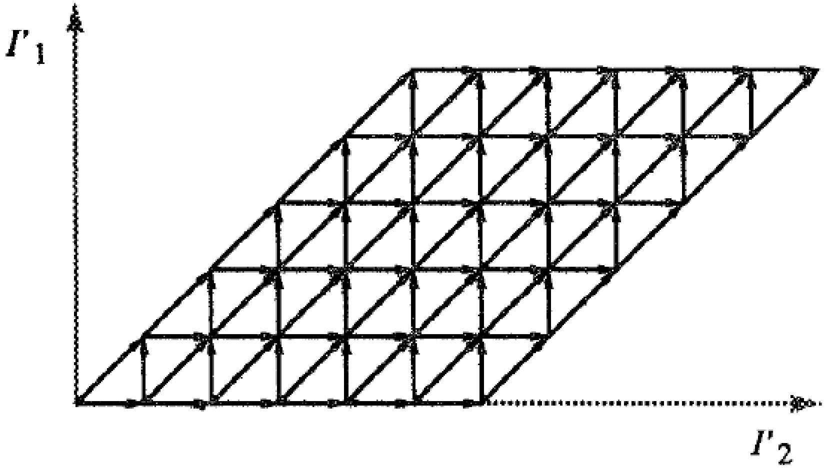
$$D = \{(0,1),(1,0),(1,-1)\}.$$



```
for $I'_1$ := 0 to 5 do
  for $I'_2$ := $I'_1$ to 6+$I'_1$ do
    A[$I'_2-I'_1+1$] := 1/3 * (A[$I'_2-I'_1$]
        + A[$I'_2-I'_1+1$] + A[$I'_2-I'_1+2$]);
```

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

$$D' = TD = \{(0,1),(1,1),(1,0)\}$$



Loop Strip Mining

Informal Definition Convert a single loop into two nested loops for a specified “block size”

(Always safe.)

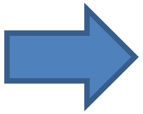
```
for i=1 to N
     A[i] = x + B[i] * 2
end for
```

```
for ii=1 to N step B
   for i=ii to min(ii+B-1, N)
       A[i] = x + B[i] * 2
   end for
end for
```

Loop Strip Mining Applications

- **Loop tiling:** ***strip-mine*** and then ***interchange*** multiple uses. Can be useful for increasing cache locality or blocking parallel loops;

```
for j=1 to N
  for ii=1 to N step B
    for i=ii to min(ii+B-1, N)
       A[i][j] = x + B[i][j]
```



```
for ii=1 to N step B
  for j=1 to N
    for i=ii to min(ii+B-1, N)
       A[i][j] = x + B[i][j]
```

When is it safe to do tiling?

- **Prefetching:** strip-mine by cache line size; prefetch once per outer iteration

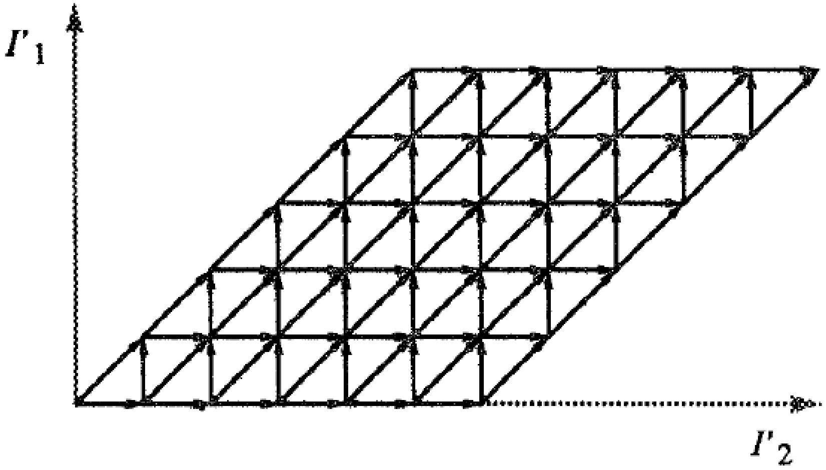
- **Instruction scheduling:** strip-mine and then unroll inner loop

Tiling Example

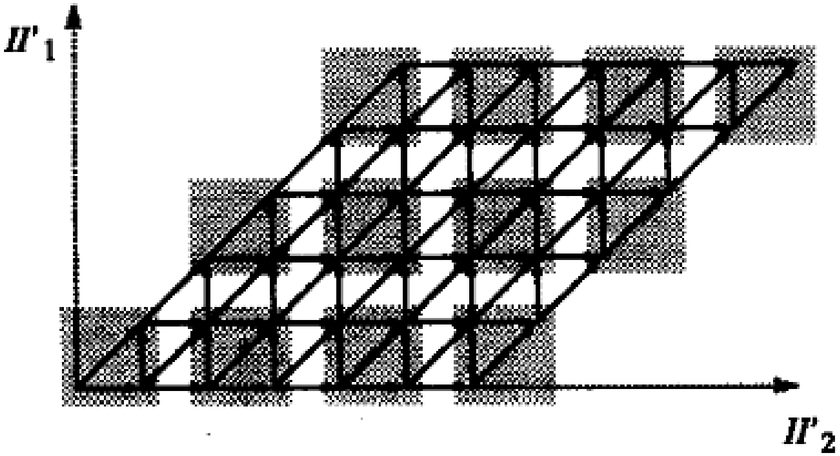
```
for $I'_1$ := 0 to 5 do
  for $I'_2$ := $I'_1$ to 6+$I'_1$ do
    A[$I'_2 - I'_1 + 1$] := 1/3 * (A[$I'_2 - I'_1$]
          + A[$I'_2 - I'_1 + 1$] + A[$I'_2 - I'_1 + 2$]);
```

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

$$D' = TD = \{(0,1),(1,1),(1,0)\}$$



```
for $II'_1$ := 0 to 5 by 2 do
  for $II'_2$ := 0 to 11 by 2 do
    for $I'_1$ := $II'_1$ to min(5,$II'_1$ + 1) do
      for $I'_2$ := max($I'_1$, $II'_2$) to min(6+$I'_1$, $II'_2$+1) do
        a[$I'_2 + 1$] := 1/3 * (a[$I'_2$] + a[$I'_2 + 1$] + a[$I'_2 + 2$]);
```



Loop Distribution

Informal Definition: Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

```
      for i = 2 to N
S1:     A[i] = B[i] + C[i]
S2:     D[i] = A[i] * 2.0
S3:     B[i+1] = A[i] * 3.0
      end for
```

```
      for i = 2 to N
S1:     A[i] = B[i] + C[i]
S3:     B[i+1] = A[i] * 3.0
      end for
      for i = 2 to N
S2:     D[i] = A[i] * 2.0
      end for
```

Loop Distribution Applications

- Create perfect loops nests for other transformations like loop interchange
- Convert a loop-carried dependence within a loop into a loop-independent dependence crossing two loops:

```
     for i=2 to N
S1:      A[i] = B[i] + C[i]
S2:      D[i] = A[i-1] * 2.0
     end for
```

```
     for i=2 to N
S1:    A[i] = B[i] + C[i]
     end for
     for i=2 to N
S2:    D[i] = A[i-1] * 2.0
     end for
```

Maximal Loop Distribution

- Identify the SCCs of the data dependence graph, to group statements in an SCC in a single loop nest
- Sort the SCCs using a topological sort on the dependence graph
- Generate distinct loop nests, one for each SCC, in sorted order
- If we have control dependence between a statement S1 is one SCC and the statement S2 in another SCC, create an array ‘flags’ that contains the Boolean conditions, populate it in the first SCC that induce dependence and use them in the second SCC.

Reminder:

- ***Strongly connected graph***: a directed graph in which there is a path between all pairs of vertices.
- ***Strongly connected component (SCC)*** is a maximal strongly connected subgraph

Loop Fusion

Informal Definition: Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

```
for i=1 to N
   A[i] = i*i
end for
for i=1 to N
   B[i] = A[i] + 1
end for
```

```
for i=1 to N
   A[i] = i*i
   B[i] = A[i] + 1
end for
```

Loop Fusion

```
for i=1 to M
  for j=1,N-1
    A[j,i] = i*i + j*j
  end for

  for j=1 to N
    B[j,i] = A[j,i] + i + j
  end for
end for
```

```
for i=1 to M
  for j=1 to N-1
    A[j,i] = i*i + j*j
    B[j,i] = A[j,i] + i + j
  end for
  // peel last iteration:
  j=N
  B[j,i] = A[j,i] + i + j
end for
```

Loop Fusion Motivation

- Increase cache reuse (if same array accessed in two loops) Fundamental optimization for array languages (e.g., Fortran 90, HPF, MATLAB, APL)

  Example in F90:

  ```
  A[1:M, 1:N] = B[1:M, 1:N] * 2
  C[1:M, 1:N] = A[1:M, 1:N] + 1
  ```

- Increase granularity of parallelism (work per iteration) Important for shared-memory parallelism (the model with parallel loop and barriers)

Legality of Loop Fusion

Fusion-Preventing Dependence: A loop-independent dependence from S1 to S2 in different loops is fusion-preventing if fusing the two loops causes the dependence to become a loop-carried dependence from S2 to S1.

Legality of Loop Fusion: Two loops can be fused if ***all three*** conditions are satisfied:

1. Both have identical bounds (*transform loops if needed*)
2. There is no fusion-preventing dependence between them.
3. There is no path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.

Loop Fusion: Illegal Cases

```
for i=1 to M
  for j=2 to N
    A[j,i] = B[j-1,i] * 2
  end for

  for j=2 to N
    B[j,i] = A[j,i] * 3
  end for
end for
```

```
for i=1 to M
  for j=2 to N
    t[j] = B[j-1,i]
  end for

  for j=2 to N
    A[j,i] = t[j] * 2
    B[j,i] = A[j,i] * 3
  end for
end for
```

Create temporary array to make fusion possible

Loop Alignment

Informal Definition: Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations

(Always safe)

```
for i=2 to N
   A[i] = B[i] + C[i]
   D[i] = A[i-1] * 2.0
end for
```

```
i = 1
D[i+1] = A[i] * 2

for i=2 to N-1
   A[i] = B[i] + C[i]
   D[i+1] = A[i] * 2.0
end for

i = N
A[i] = B[i] + C[i]
```

Scalar Replacement

Informal Definition: Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```
for i = 1 to n
   for j = 2 to n
      x[j,i] = a[i] +
               x[j-1,i] +
               b[j,i]
   end for
end for
```

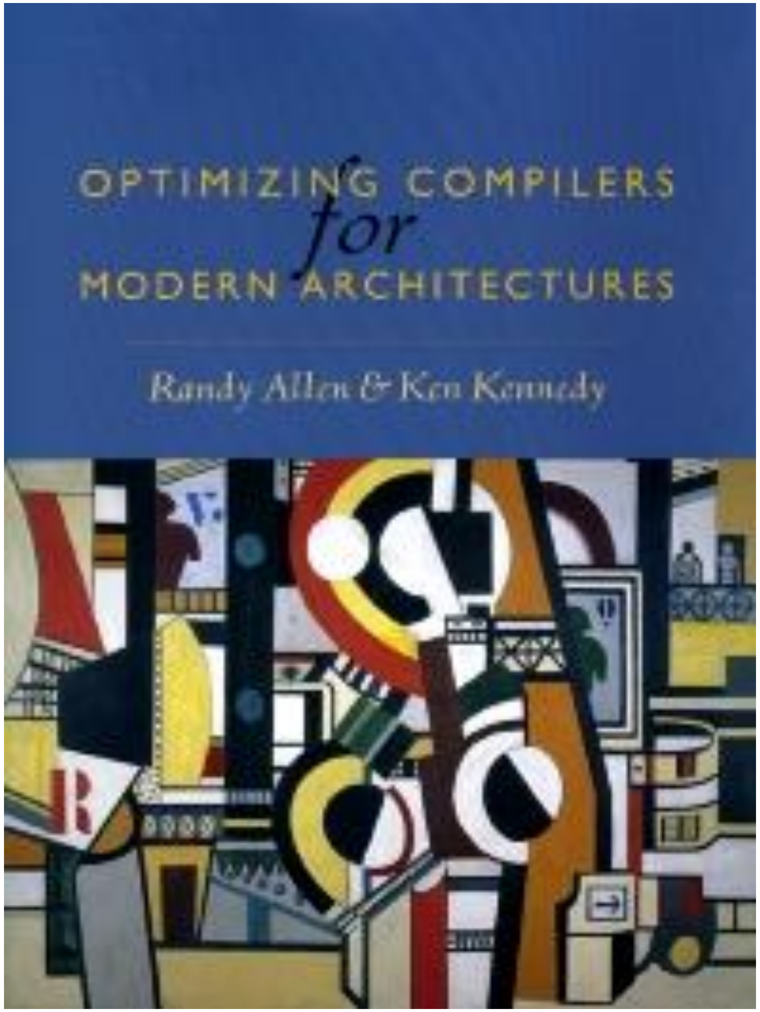
```
for i = 1 to n
   t1 = a[i];
   for j = 2 to n
      x[j,i] = t1 +
               x[j-1,i] +
               b[j,i]
   end for
end for
```

Unroll and Jam

Informal Definition: Unroll the outer loop by k, then fuse the resulting k inner loops into a single loop

```
for i = 1 to n
  for j = 1 to n
    a[i] = a[i] + b[j]
  end for
end for
```

```
for i = 1 to n step 2
  for j = 1 to n
    a[i] = a[i] + b[j]
    a[i+1] = a[i+1] + b[j]
  end for
end for
```



More details:

Optimizing Compilers for Modern Architectures

Allen and Kennedy

Academic Press