CS 610: Loop Transformations

Swarnendu Biswas

Semester 2023-24-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Enhancing Program Performance

Possible ideas

- Adequate fine-grained parallelism
  - Multiple pipelined functional units in each core
  - Exploit vector instruction sets (SSE, AVX, AVX-512)
- Adequate parallelism for SMP-type systems
  - Keep multiple asynchronous processors busy with work
- Minimize cost of memory accesses

Role of a Good Compiler

Try and extract performance automatically

Optimize memory access latency

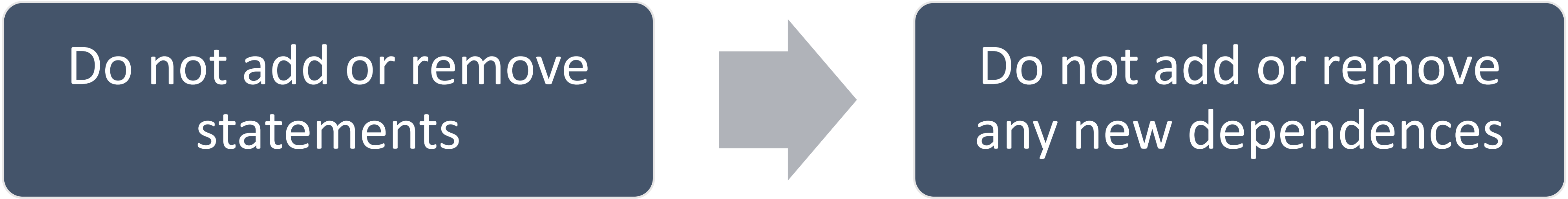
- **Code restructuring** optimizations
- Prefetching optimizations
- Data layout optimizations
- Code layout optimizations

Loop Optimizations

- Loops are one of most commonly used constructs in HPC program
- Compiler performs many loop optimization techniques automatically
  - In some cases, source code modifications can enhance optimizer’s ability to transform code

Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop



Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop

A reordering transformation is valid if it preserves all existing dependences in the loop

Iteration Reordering and Parallelization

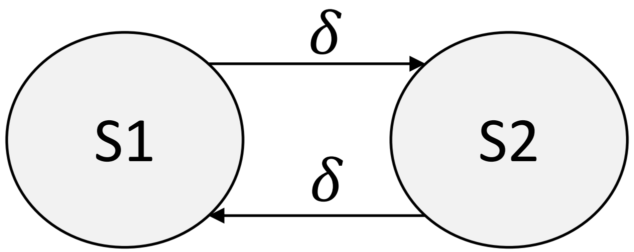
- A transformation that reorders the iterations of a level-*k* loop, without making any other changes, is valid if the loop carries no dependence

- Each iteration of a loop may be executed in parallel if it carries no dependences

Data Dependence Graph and Parallelization

- If the Data Dependence Graph (DDG) is acyclic, then vectorization of the program is possible and is straightforward
- Otherwise, try to transform the DDG to an acyclic graph

```
      FOR I=1,N
        FOR J=1,M
S1        A(I,J) = B(I-1,J+1) + C
S2        B(I,J) = A(I-1,J-1) + K
```



Enhancing Fine-Grained Parallelism

Focus is on vectorization of inner loops

System Setup

- Setup: vector or superscalar architectures

- Focus is mostly on parallelizing the inner loops
- We will see optimizations for coarse-grained parallelism later

Loop Interchange (Loop Permutation)

- Switch the nesting order of loops in a **perfect** loop nest
- Can increase parallelism, can improve spatial locality

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

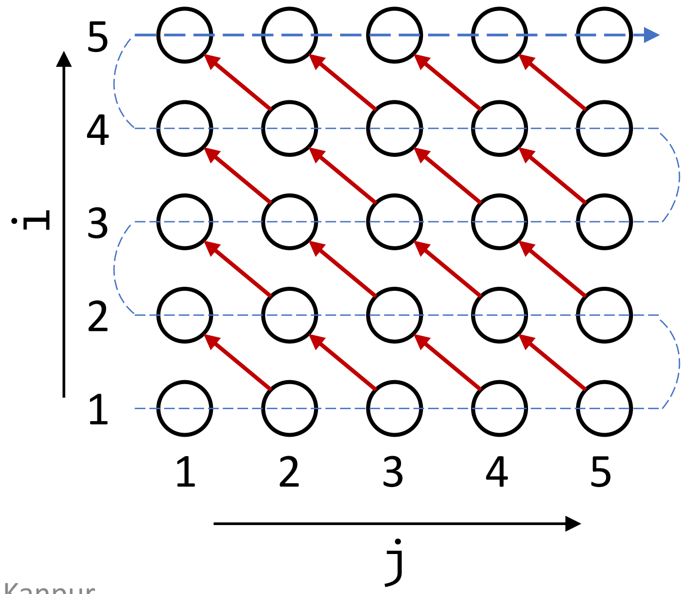
- Dependence is now carried by the outer loop, inner loop can be vectorized

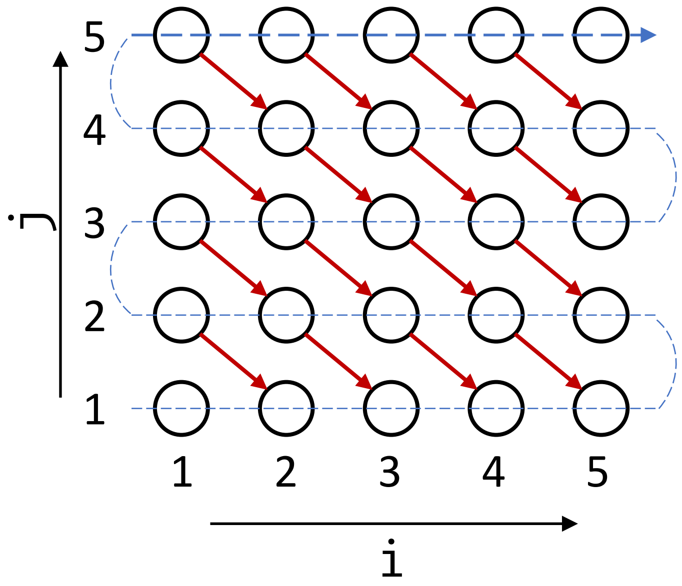
```
  DO J = 1, M
    DO I = 1, N
S     A(I,J+1) = A(I,J) + B
    ENDDO
  ENDDO
```

Example of Loop Interchange

```
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
```





Validity of Loop Interchange

1. Construct direction vectors for all possible dependences in the loop to form a direction matrix
   - Identical direction vectors are represented by a single row in the matrix
2. Compute direction vectors after permutation
3. Permutation of the loops in a perfect nest is legal iff there are no “-” direction as the leftmost non–“0” direction in any direction vector

Validity of Loop Interchange

- Loop interchange is valid for a 2D loop nest if none of the dependence vectors has any negative components
- Interchange is legal: (1,1), (2,1), (0,1), (3,0)
- Interchange is not legal: (1,-1), (3,-2)

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

Validity of Loop Permutation

- Generalization to higher-dimensional loops: Permute all dependence vectors exactly the same way as the intended loop permutation
- If any permuted vector is lexicographically negative, permutation is illegal
- Example: $d_1 = (1,-1,1)$ and $d_2 = (0,2,-1)$
  - `ijk` → `jik`? (1,-1,1) → (-1,1,1): illegal
  - `ijk` → `kij`? (0,2,-1) → (-1,0,2): illegal
  - `ijk` → `ikj`? (0,2,-1) → (0,-1,2): illegal
  - No valid permutation: `j` cannot be outermost loop (-1 component in d_1), and `k` cannot be outermost loop (-1 component in d_2)
- A loop nest is **fully** permutable if any permutation transformation to the loop nest is legal

Benefits from Loop Permutation

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

Stride	ikj	kij	jik	ijk	jki	kji
C[i][j]	1	1	0	0	n	n
A[i][k]	0	0	1	1	n	n
B[k][j]	1	1	n	n	0	0

Understanding Loop Interchange

Pros	Cons
• Goal is to improve locality of reference or allow vectorization	• Need to be careful about the iteration order, order of array accesses, and data involved

Does Loop Interchange/Permutation Always Help?

```
do i = 1, 10000
   do j = 1, 1000
     a[i] = a[i] + b[j,i] * c[i]
   end do
 end do
```

```
do I = 1, N
   do J = 1, M
     do K = 1, L
       A(I+1,J+1,K) = A(I,J,K) + B
     end do
   end do
end do
```

- Type and benefit from loop interchange depends on the target machine, the data structures accessed, memory layout and stride patterns
- Optimization choices for the snippet on the right: vectorize `J` and `K`, parallelize `K` with threads, and vectorize `I` assuming column-major layout

Loop Shifting

- In a perfect loop nest, if loops at level $i, i+1, \ldots, i+n$ carry no dependence, i.e., all dependences are carried by loops at level less than i or greater than $i+n$, then it is always legal to shift these loops inside of loop $i+n+1$.
- These loops will not carry any dependences in their new position.

Loops i to $i+n$

Dependence carried by outer loops	+	0	+	0	0	0	
	0	+	-	+	+	0	
	0	0	0	0	+	+	Dependence carried by inner loops
	0	0	0	0	0	+	

Loop Shift for Matrix Multiply

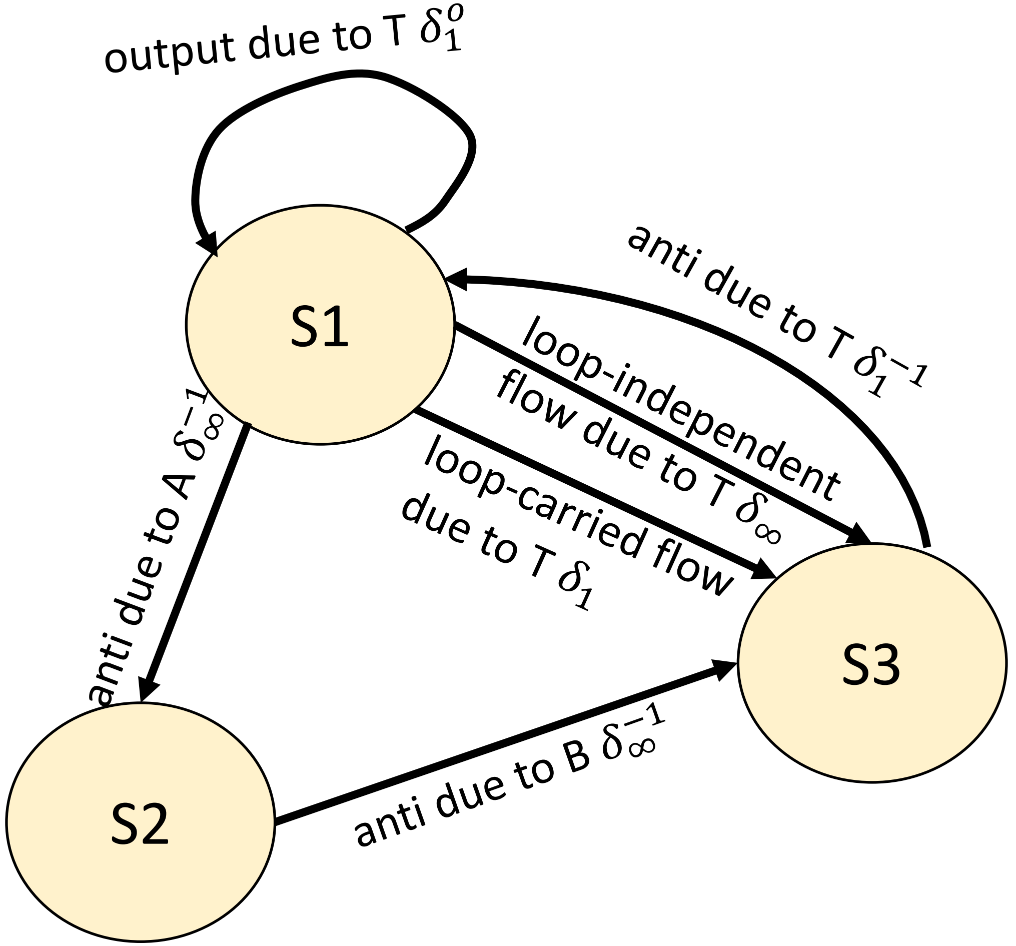
```
    DO I = 1, N
      DO J = 1, N
        DO K = 1, N
S         A(I,J) = A(I,J) + B(I,K)*C(K,J)
        ENDDO
      ENDDO
    ENDDO
```

Is the loop nest vectorizable as is?

We can move loops I and J inside

Scalar Expansion

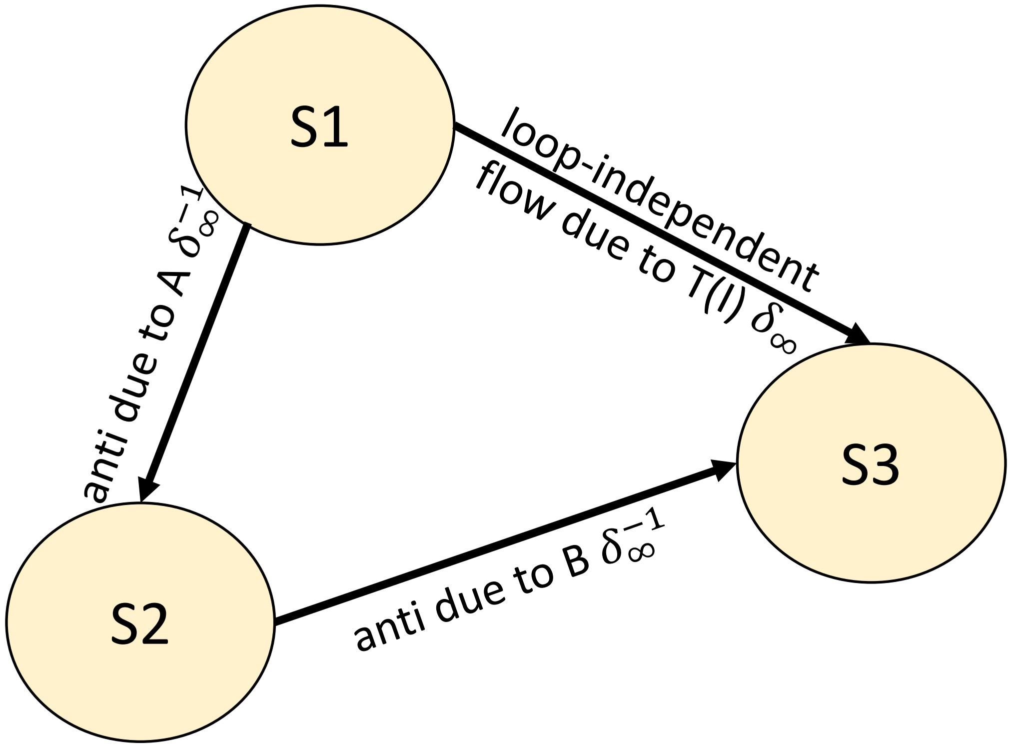
```
    DO I = 1, N
S1    T = A(I)
S2    A(I) = B(I)
S3    B(I) = T
    ENDDO
```



Scalar Expansion

```
    DO I = 1, N
S1    $T(I) = A(I)
S2    A(I) = B(I)
S3    B(I) = $T(I)
    ENDDO
    T = $T(N)
```

Eliminates dependences that arise from reuse of memory locations at the cost of extra memory



Scalar Expansion

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

```
$T(0) = T
DO I = 1, N
  $T(I) = $T(I-1) + A(I) + A(I-1)
  A(I) = $T(I)
ENDDO
T = $T(N)
```

Can we parallelize the I loop?
Check the dependence graph.

Understanding Scalar Expansion

Pros	Cons
• Eliminates dependences due to reuse of memory locations, helps with parallelism	• Increases memory overhead • Incurs addressing overhead

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

Can also try forward substitution

➡

Strip-mining

```
DO I = 1, N, 64
  DO i = 0, 63
    T = A(I+i) + A(I+1+i)
    A(I+i) = T + B(I+i)
ENDDO
```

➡

Strip loop

```
DO I = 1, N, 64
  DO i = 0, 63
    $T(i) = A(I+i) + A(I+1+i)
    A(I+i) = $T(i) + B(I+i)
ENDDO
```

Limits of Scalar Expansion

```
    DO I = 1, 100
S1    T = A(I) + B(I)
S2    C(I) = T + T
S3    T = D(I) - B(I)
S4    A(I+1) = T * T
    ENDDO
```

```
    DO I = 1, 100
S1    $T(I) = A(I) + B(I)
S2    C(I) = $T(I) + $T(I)
S3    $T(I) = D(I) - B(I)
S4    A(I+1) = $T(I) * $T(I)
    ENDDO
```

Can we vectorize this loop nest? Check the dependence graph.

Scalar Renaming

```
      DO I = 1, 100
S1      T = A(I) + B(I)
S2      C(I) = T + T
S3      T = D(I) - B(I)
S4      A(I+1) = T * T
      ENDDO
```

```
      DO I = 1, 100
S1      T1 = A(I) + B(I)
S2      C(I) = T1 + T1
S3      T2 = D(I) - B(I)
S4      A(I+1) = T2 * T2
      ENDDO
      T = T2
```

Can we vectorize this loop nest as is?

Allow Vectorization with Statement Interchange

```
    DO I = 1, 100
S1    T1 = A(I) + B(I)
S2    C(I) = T1 + T1
S3    T2 = D(I) - B(I)
S4    A(I+1) = T2 * T2
    ENDDO
    T = T2
```

➡

```
    DO I = 1, 100
S3    T2 = D(I) - B(I)
S4    A(I+1) = T2 * T2
S1    T1 = A(I) + B(I)
S2    C(I) = T1 + T1
    ENDDO
    T = T2
```

➡

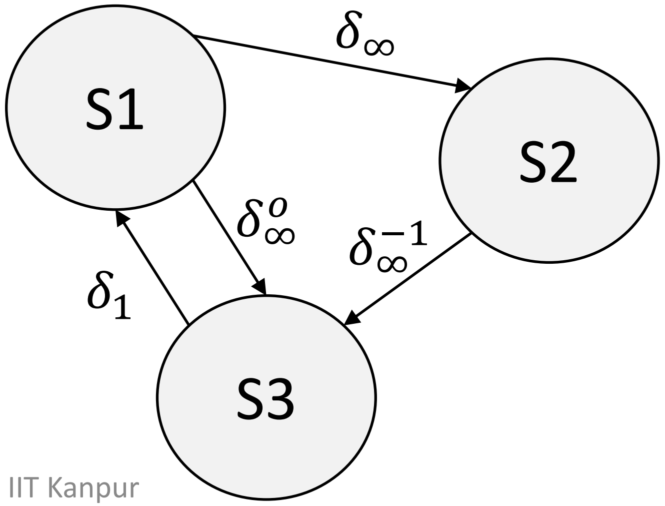
```
S3    T2[1:100] = D(1:100) - B(1:100)
S4    A[2:101] = T2[1:100] * T2[1:100]
S1    T1[1:100] = A[1:100] + B[1:100]
S2    C[1:100] = T1[1:100] + T1[1:100]

    T = T2[100]
```

Array Renaming

```
    DO I = 1, 100
S1     A(I) = A(I-1) + X
S2     Y(I) = A(I) + Z
S3     A(I) = B(I) + C
    ENDDO
```

```
    DO I = 1, 100
S1     $A(I) = A(I-1) + X
S2     Y(I) = $A(I) + Z
S3     A(I) = B(I) + C
    ENDDO
```

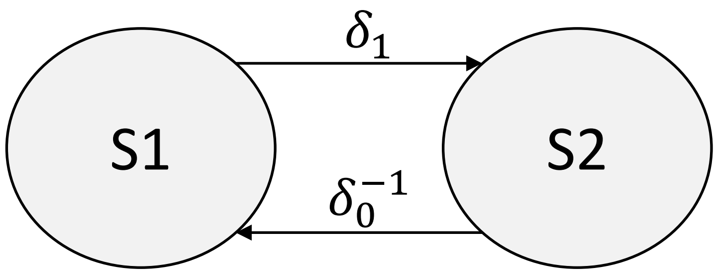


Array renaming requires sophisticated analysis

Node Splitting

```
    DO I = 1, 100
S1    A(I) = X(I+1) + X(I)
S2    X(I+1) = B(I) + 10
    ENDDO
```

```
    DO I = 1, 100
S0    $X(I) = X(I+1)
S1    A(I) = $X(I) + X(I)
S2    X(I+1) = B(I) + 10
    ENDDO
```



Index-Set Splitting

```
DO I = 1, 100
  A(I+20) = A(I) + B
ENDDO
```

```
DO I = 1, 100, 20
  DO i = I, I+19
    A(i+20) = A(i) + B
  ENDDO
ENDDO
```

strip-mining

An index-set splitting transformation subdivides the loop into different iteration ranges

Loop Peeling

- Splits any problematic iterations (could be first, middle, or last few) from the loop body
- Change from a loop-carried dependence to loop-independent dependence
- Transformed loop carries no dependence, can be parallelized
- Peeled iterations execute in the original order, transformation is always legal to perform

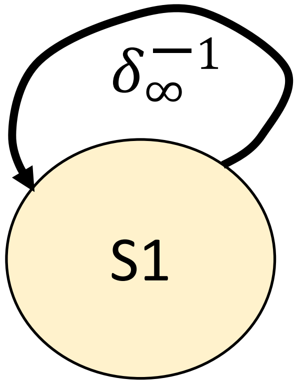
```
DO I = 1, N
  A(I) = A(I) + A(1)
ENDDO
```

```
A(1) = A(1) + A(1)
DO I = 2, N
  A(I) = A(I) + A(1)
ENDDO
```

Loop splitting

Loop Splitting

```
DO I = 1, N
  A(I) = A(N/2) + B(I)
ENDDO
```



assume N is divisible by 2

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

Section-Based Splitting

```
     DO I = 1,N
       DO J = 1, N/2
S1       B(J,I) = A(J,I) + C
  δ1   ENDDO
       DO J = 1,N        δ∞
S2       A(J,I+1) = B(J,I) + D
       ENDDO
     ENDDO
```

```
     DO I = 1,N
       DO J = 1, N/2
S1       B(J,I) = A(J,I) + C
       ENDDO
       DO J = 1,N/2
S2       A(J,I+1) = B(J,I) + D
       ENDDO
       DO J = N/2+1, N
S3       A(J,I+1) = B(J,I) + D
       ENDDO
     ENDDO
```

S3 is independent

Enabling Vectorization with Section-Based Splitting

```
    DO I = 1,N
      DO J = 1, N/2
S1      B(J,I) = A(J,I) + C
      ENDDO
      DO J = 1,N/2
S2      A(J,I+1) = B(J,I) + D
      ENDDO
      DO J = N/2+1, N
S3      A(J,I+1) = B(J,I) + D
      ENDDO
    ENDDO
```

```
    DO I = 1,N
      DO J = N/2+1, N
S3      A(J,I+1) = B(J,I) + D
      ENDDO
    DO I = 1,N
      DO J = 1,N/2
S1      B(J,I) = A(J,I) + C
      ENDDO
      DO J = 1, N/2
S2      A(J,I+1) = B(J,I) + D
      ENDDO
    ENDDO
```

Enabling Vectorization with Section-Based Splitting

```
    DO I = 1,N
      DO J = N/2+1, N
S3      A(J,I+1) = B(J,I) + D
      ENDDO
    DO I = 1,N
      DO J = 1,N/2
S1      B(J,I) = A(J,I) + C
      ENDDO
      DO J = 1, N/2
S2      A(J,I+1) = B(J,I) + D
      ENDDO
    ENDDO
```

cannot vectorize I

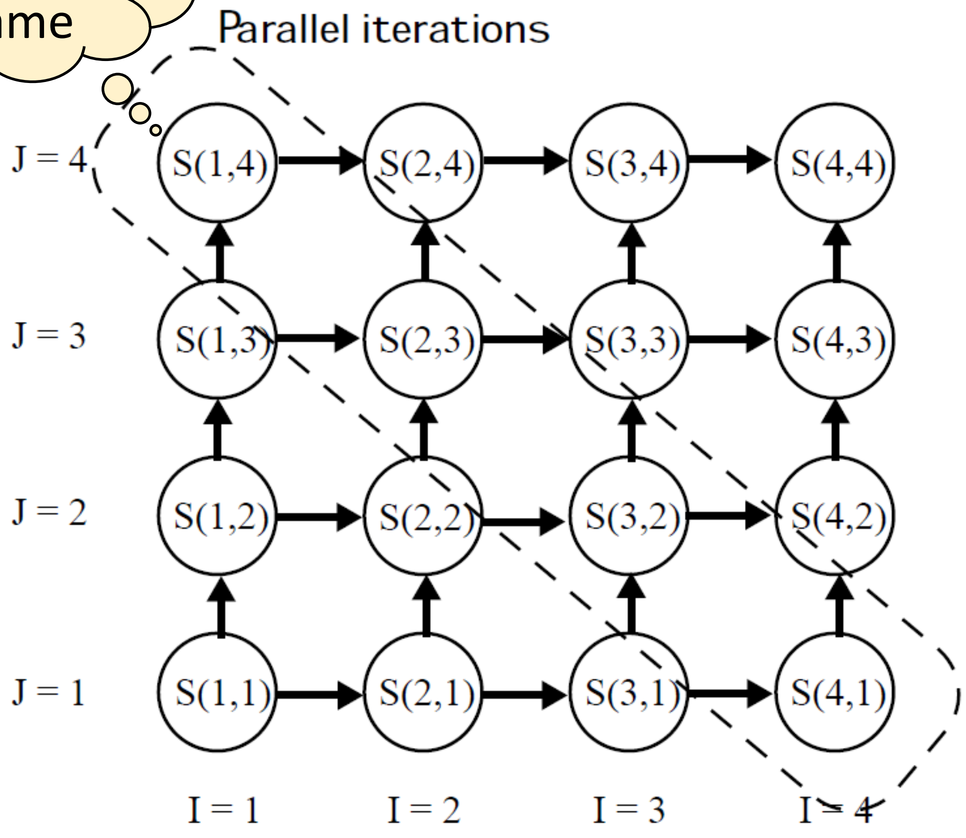
```
    M = N/2
S3  A(M+1:N,2:N+1) = B(M+1:N,1:N) + D
    DO I = 1, N
S1    B(1:M,I) = A(1:M,I) + C
S2    A(1:M,I+1) = B(1:M,I) + D
    ENDDO
```

Loop Skewing

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

Which loops carry dependences?

I+J is same

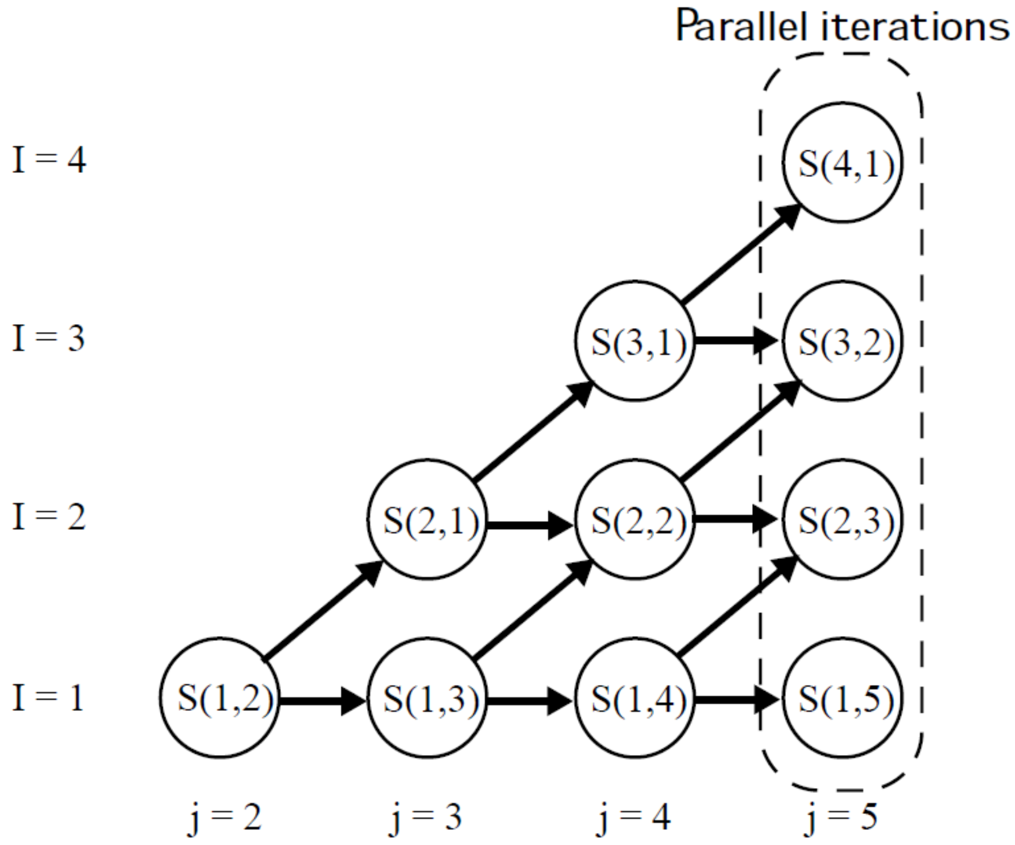


Loop Skewing

j = I+J

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

What are the dependences now? Which loop carries the dependence?



Loop skewing skews the execution of the inner loop relative to the outer loop by adding the index of the outer loop times a skewing factor f to the bounds of the inner loop and subtracting the same value from all the uses of the inner loop index.

Perform Loop Interchange

Given a dependency vector (a, b), skewing transforms it to $(a, fa + b)$.

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

Are the loops vectorizable?

⬇

```
    DO j = 2, N+N
      DO I = max(1,j-N), min(N,j-1)
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
      ENDDO
    ENDDO
```

Can use Fourier-Motzkin elimination

Understanding Loop Skewing

Pros	Cons
• Reshapes the iteration space to find possible parallelism • Preserves lexicographic order of the dependences, is always legal • Allows for loop interchange in future	• Resulting iteration space can be trapezoidal • Irregular loops are not very amenable for vectorization • Need to be careful about load imbalance

Loop Unrolling (Loop Unwinding)

- Reduce number of iterations of loops
- Add statement(s) to do work of missing iterations
- JIT compilers try to perform unrolling at run-time

4-way inner loop unrolling

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    y[i] = y[i] + a[i][j]*x[j];
  }
}
```

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j+=4) {
    y[i] = y[i] + a[i][j]*x[j]
                + a[i][j+1]*x[j+1]
                + a[i][j+2]*x[j+2]
                + a[i][j+3]*x[j+3];
  }
}
```

Outer Loop Unrolling + Inner Loop Jamming

```
for (i=0; i<2*n; i++) {
  for (j=0; j<m; j++) {
    loop-body(i,j);
  }
}
```

```
for (i=0; i<2*n; i+=2) {
  for (j=0; j<m; j++) {
    loop-body(i,j)
  }
  for (j=0; j<m; j++) {
    loop-body(i+1,j)
  }
}
```

```
for (i=0; i<2*n; i+=2) {
  for (j=0; j<m; j++) {
    loop-body(i,j)
    loop-body(i+1,j)
  }
}
```

2-way outer unroll does not increase operation-level parallelism in the inner loop

Is Unroll and Jam Legal?

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

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

Validity Condition for Loop Unroll/Jam

- Complete unroll/jam of a loop is equivalent to a loop permutation that moves that loop innermost, without changing order of other loops
- If such a loop permutation is valid, unroll/jam of the loop is valid
- Example: 4D loop `ijkl`; d_1 = (1,-1,0,2), d_2 = (1,1,-2,-1)
  - `i`: d_1 $\rightarrow$ (-1,0,2,**1**) $\Longrightarrow$ invalid to unroll/jam
  - `j`: d_1 $\rightarrow$ (1,0,2**,-1**); d2 -> (1,-2,-1,**1**) $\Longrightarrow$ valid to unroll/jam
  - `k`: d_1 $\rightarrow$ (1,-1,2,**0**); d2 -> (1,1,-1,**-2**) $\Longrightarrow$ valid to unroll/jam
  - `l`: d_1 and d_2 are unchanged; innermost loop always unrollable

Understanding Loop Unrolling

Pros	Cons
• Small loop bodies are problematic, reduces control overhead of loops • Increases operation-level parallelism in loop body • Allows other optimizations like reuse of temporaries across iterations	• Increases the executable size • Increases register usage • May prevent function inlining

Loop Tiling (Loop Blocking)

- Improve data reuse by chunking the data in to smaller tiles (blocks)
  - All the required blocks are supposed to fit in the cache
- Performs strip mining in multiple array dimensions
- Tries to exploit spatial and temporal locality of data

```
for (i = 0; i < N; i++) {
  …
}
```

```
for (j = 0; j < N; j +=B) {
  for (i = j; i < min(N, j+B); j++) {
    …
  }
}
```

MVM with 2x2 Blocking

```
int i, j, n = 100;
int a[100][100], b[100], c[100];
for (i=0; i<n; i++) {
  c[i] = 0;
  for (j=0; j<n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

```
int i, j, x, y, n = 100;
int a[100][100], b[100], c[100];
for (i=0; i<n; i+=2) {
  c[i] = 0;
  c[i + 1] = 0;
  for (j=0; j<n; j+=2) {
    for (x=i; x<min(i+2,n); x++) {
      for (y=j; y<min(j+2,n); y++)
        c[x] = c[x] + a[x][y] * b[y];
    }
  }
}
```

Loop nest optimization

Loop Tiling

- Determining the tile size
  - Requires accurate estimate of array accesses and the cache size of the target machine
  - Loop nest order also influences performance
  - Difficult theoretical problem, usually heuristics are applied
  - Cache-oblivious algorithms make efficient use of cache without explicit blocking

Validity Condition for Loop Tiling

- A band of loops is fully permutable if all permutations of the loops in that band are legal
- A contiguous band of loops can be tiled if they are fully permutable
- Example: d = (1,2,-3)
  - Tiling all three loops `ijk` is not valid, since the permutation `kij` is invalid
  - 2D tiling of band `ij` is valid
  - 2D tiling of band `jk` is valid

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      loop_body(i,j,k)
```

⬇

```
for (it = 0; it < n; it+=T)
  for (jt = 0; tj < n; j+=T)
    for (i = it; i < it+T; i++)
      for (j = jt; j < jt+T; j++)
        for (k = 0; k < n; k++)
          loop_body(i,j,k)
```

Creating Coarse-Grained Parallelism

Focus is on parallelism of outer loops

Find Work For Threads

- Setup
  - Symmetric multiprocessors with shared memory
  - Threads are running on each core and are coordinating execution with occasional synchronization
    - A basic synchronization element is a barrier
    - A barrier in a program forces all processes to reach a certain point before execution continues

- Challenge: Balance the granularity of parallelism with communication overheads

Challenges in Coarse-Grained Parallelism

Minimize communication and synchronization overhead while evenly load balancing across the processors

- Running everything on one processor achieves minimal communication and synchronization overhead
- Very fine-grained parallelism achieves good load balance, but benefits may be outweighed by frequent communication and synchronization

Challenges in Coarse-Grained Parallelism

Minimize communication and synchronization overhead while evenly load balancing ac

- Running … n achieves achieves m … nefits may synchroniz … requent … and synchronization

One expectation from an optimizing compiler is to find the sweet spot

Few Ideas to Try

- Single loop
  - Carries a dependence $\Longrightarrow$ Try transformations (e.g., loop distribution and scalar expansion) to eliminate the loop-carried dependence
  - Decide on the granularity of the new parallel loop
- Perfect loop nests
  - Try loop interchange to see if the dependence level can be changed

Privatization

- Privatization is similar in flavor to scalar expansion
- Temporaries can be made local to each iteration

```
    DO I = 1,N
S1     T = A(I)
S2     A(I) = B(I)
S3     B(I) = T
    ENDDO
```

```
    PARALLEL DO I = 1,N
      PRIVATE t
S1    t = A(I)
S2    A(I) = B(I)
S3    B(I) = t
    ENDDO
```

Privatization

- A scalar variable x in a loop L is **privatizable** if every path from the entry of L to a use of x in the loop passes through a definition of x
  - No use of the variable is upward exposed, i.e., the use never reads a value that was assigned outside the loop
  - No use of the variable is from an assignment in an earlier iteration
- Computing upward-exposed variables from a block x

$$up(x) = use(x) \cup (\neg\, def(x) \cap \bigcup_{y \in succ(x)} up(y))$$

- Computing privatizable variables for a loop body B where b_0 is the entry block

$$private(B) = \neg up(b_0) \cap (\bigcup_{y \in B} def(y))$$

Privatization

- If all dependences carried by a loop involve a privatizable variable, then loop can be parallelized by making the variables private

- Preferred compared to scalar expansion
  - Less memory requirement
  - Scalar expansion may suffer from **false sharing**

- However, there can be situations where scalar expansion works but privatization does not

Comparing Privatization and Scalar Expansion

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

Scalar ➡ Expansion

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

δ_1^{-1} δ_∞

⬇ Privatization

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

⬇

```
PARALLEL DO I = 1, N
  T$(I) = A(I) + B(I)
ENDDO

PARALLEL DO I = 1, N
  A(I-1) = T$(I)
ENDDO
```

Loop Distribution (Loop Fission)

```
    DO I = 1, 100
      DO J = 1, 100
S1      A(I,J) = B(I,J) + C(I,J)
S2      D(I,J) = A(I,J-1) * 2.0
      ENDDO
    ENDDO
```

δ

```
    DO I = 1, 100
      DO J = 1, 100
S1      A(I,J) = B(I,J) + C(I,J)
      ENDDO

      DO J = 1, 100
S2      D(I,J) = A(I,J-1) * 2.0
      ENDDO
    ENDDO
```

Eliminates loop-carried dependences

Validity Condition for Loop Distribution

- Sufficient (but not necessary) condition: A loop with two statements can be distributed if there are no dependences from any instance of the **later** statement to any instance of the **earlier** one
  - Generalizes to more statements

```
    For I = 1, N
S1    A(I) = B(I) + C(I)
S2    E(I) = A(I+1) * D(I)
    EndFor
```

δ_1^{-1}

```
    For I = 1, N
S1    A(I) = B(I) + C(I)
S2    E(I) = A(I-1) * D(I)
    EndFor
```

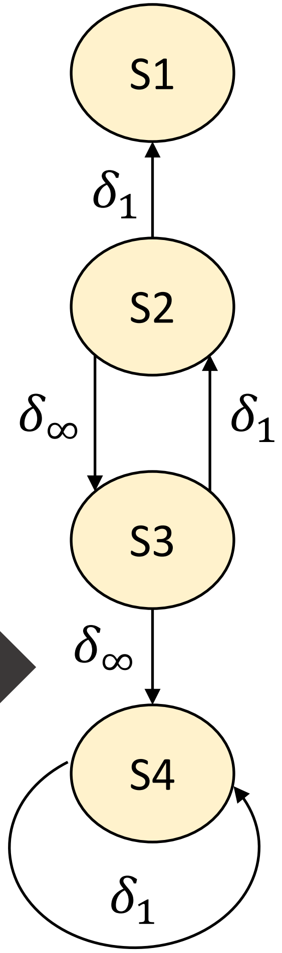
δ_1

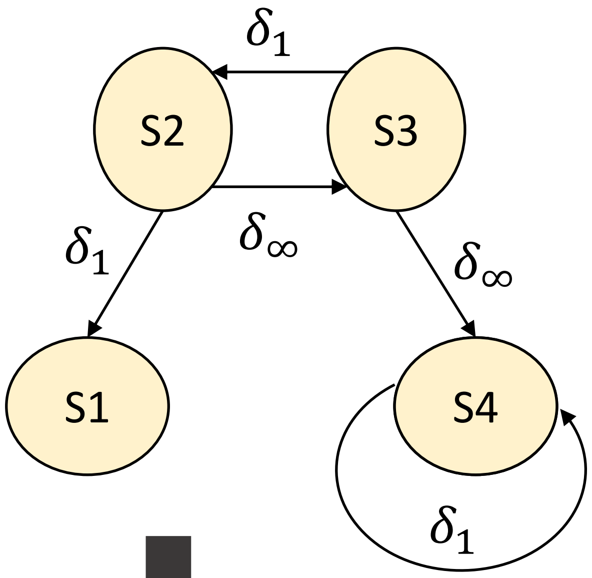
Performing Loop Distribution

Steps

i. Build the DDG
ii. Identify strongly-connected components (SCCs) in the DDG
iii. Make each SCC a separate loop
iv. Arrange the new loops in a topological order of the DDG

```
    FOR I = 1, N
S1:   A[I] = A[I] + B[I-1]
S2:   B[I] = C[I-1] + X
S3:   C[I] = B[I] + Y
S4:   D[I] = C[I] + D[I-1]
```






```
    FOR I = 1, N
S2:   B[I] = C[I-1] + X
S3:   C[I] = B[I] + Y
    FOR I = 1, N
S1:   A[I] = A[I] + B[I-1]
    FOR I = 1, N
S4:   D[I] = C[I] + D[I-1]
```

Understanding Loop Distribution

Pros	Cons
• Execute source of a dependence **before** the sink • Reduces the memory footprint of the original loop for both data and code • Improves opportunities for vectorization	• Can increase the synchronization required between dependence points

Loop Alignment

- Unlike loop distribution, realign the loop to compute and use the values in the same iteration

```
    DO I = 2, N
S1    A(I) = B(I) + C(I)
S2    D(I) = A(I-1) * 2.0
    ENDDO
```

Cannot be parallelized

```
    DO i = 1, N+1
      if i > 1 && i < N+1
S1      A(i) = B(i) + C(i)
      if i < N
S2      D(i+1) = A(i) * 2.0
    ENDDO
```

carried dependence becomes loop independent

Can Loop Alignment Eliminate All Carried Dependences?

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    B(I+1) = A(I) + D
    ENDDO
```

A is aligned, B is misaligned

```
    DO i = 1, N+1
      if i > 1
S1      B(i) = A(i-1) + D
      if i < N+1
S2      A(i) = B(i) + C
    ENDDO
```

```
    DO I = 1, N
S1    A(I+1) = B(I) + C
S2    X(I) = A(I+1) + A(I)
    ENDDO
```

```
    DO i = 0, N
      if i > 0
S1      A(i+1) = B(i) + C
      if i < N
S2      X(i+1) = A(i+2) + A(i+1)
    ENDDO
```

Loop Fusion (Loop Jamming)

```
    DO I = 1, N
S1    A(I) = B(I) + 1
S2    C(I) = A(I) + C(I-1)
S3    D(I) = A(I) + X
    ENDDO
```

loop-carried dependence

```
L1  DO I = 1, N
      A(I) = B(I) + 1
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
L3  DO I = 1, N
      D(I) = A(I) + X
    ENDDO
```

Loop Fusion (Loop Jamming)

```
L1  DO I = 1, N
      A(I) = B(I) + 1
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
L3  DO I = 1, N
      D(I) = A(I) + X
    ENDDO
```

```
L13 PARALLEL DO I = 1, N
      A(I) = B(I) + 1
      D(I) = A(I) + X
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
```

Validity Condition for Loop Fusion

- Consider a loop-independent dependence between statements in two different loops (i.e., from S1 to S2)
- A dependence is fusion-preventing if fusing the two loops causes the dependence to be carried by the combined loop in the reverse direction (from S2 to S1)

```
    DO I = 1, N
S1    A(I) = B(I) + C
    ENDDO
    DO I = 1, N
S2    D(I) = A(I+1) + E
    ENDDO
```

Loop independent flow dependence

➡

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    D(I) = A(I+1) + E
    ENDDO
```

Backward loop-carried anti dependence

No

Understanding Loop Fusion

Pros	Cons
• Reduce overhead of loops • May improve temporal locality	• May decrease data locality in the fused loop

```
      DO I = 1, N
S1      A(I) = B(I) + C
      ENDDO
      DO I = 1, N
S2      D(I) = A(I-1) + E
      ENDDO
```

➡

```
      DO I = 1, N
S1      A(I) = B(I) + C
S2      D(I) = A(I-1) + E
      ENDDO
```

Yes

Loop Interchange

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

Loop I carries a dependence

Parallelizing J is good for vectorization, but not for coarse-grained parallelism

Loop Interchange

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

➡

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

⬇

```
PARALLEL DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
END PARALLEL DO
```

Dependence-free loops should move to the outermost level

Loop Interchange

Vectorization	Coarse-grained Parallelism
• Move dependence-free loops to innermost level	• Move dependence-free loops to outermost level

Condition for Loop Interchange

- In a perfect loop nest, a loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only “0” entries

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

Code Generation Strategy

1. Continue till there are no more columns to move
    - Choose a loop from the direction matrix that has all “0” entries in the column
    - Move it to the outermost position
    - Eliminate the column from the direction matrix
2. Pick loop with **most “+” entries**, move to the next outermost position
    - Generate a sequential loop
    - Eliminate the column
    - Eliminate any rows that represent dependences carried by this loop
3. Repeat from Step 1

Code Generation Example

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  ENDDO
ENDDO
```

What is the direction matrix? Can we permute the loops?

Code Generation Example

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  ENDDO
ENDDO
```

+	**0**	**0**
0	**0**	**+**
+	**+**	**+**

Since there are no columns with all "0" entries, none of the loops can be parallelized at the outermost level

Generated Code

```
DO I = 1, N
  PARALLEL DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  END PARALLEL DO
ENDDO
```

How did we pick loop J?

How can we parallelize this loop?

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

0	**+**	**-**
+	**0**	**-**

No single loop carries all the dependences, so we can only parallelize loop K

Loop Reversal

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

0	+	-
+	0	-

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

0	+	+
+	0	+

Loop Reversal

- When the iteration space of a loop is reversed, the direction of dependences within that reversed iteration space are also reversed
  - A “+" dependence becomes a “-" dependence, and vice versa
- We cannot perform loop reversal if the loop carries a dependence

Perform Interchange after Loop Reversal

Increases options for performing other optimizations

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

Parallelize loops I and J

+	0	+
+	+	0

Which Transformations are Most Important?

- Selecting the best loops for parallelization is a NP-complete problem
- Flow dependences are difficult to remove
  - Try to reorder statements as in loop peeling, loop distribution
- Techniques like scalar expansion, privatization can be useful
  - Loops often use scalars for temporary values

+	+	0	0
+	0	+	0
+	0	0	+
0	+	0	0
0	0	+	0
0	0	0	+

Carries most dependences!??

Unimodular Transformations

Challenges in Applying Transformations

- We have discussed transformations (legality and benefits) in isolation
- Compilers need to apply compound transformations (e.g., loop interchange followed by reversal)
- It is challenging to decide on the desired transformations and their order of application
  - Choice and order is sensitive to the program input, *a priori* order does not work

Unimodular Transformations

- A unimodular matrix is a square integer matrix having determinant 1 or -1 (e.g., $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$)
- Few loop transformations can be modelled as matrix transformations involving unimodular matrices
  - **Loop interchange** maps iteration (i, j) to iteration (j, i)

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

  - Given the transformation T is linear, the transformed dependence is given by $T\boldsymbol{d}$ where $\boldsymbol{d}$ is the dependence vector in the original iteration space

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$$

M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. TPDS'91.

Unimodular Transformations

- The transformation matrix for **loop reversal** of the outer loop `I` in a 2D loop nest is $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
- The transformation matrix for **loop skewing** of a 2D loop nest (i, j) is the identity matrix T with $T_{j,i}$ equal to f, where we skew loop j with respect to loop i by a factor f

Example of Loop Skewing

Original

```
FOR I=1,5
  FOR J=1,5
    A(I,J) = A(I-1,J) + A(I,J-1)
```

Dependences $D = \{(1,0),(0,1)\}$

Skewed

```
FOR I=1,5
  FOR j=I+1,I+5
    A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
```

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

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

Representing Compound Transformations

```
for I = 1, N
  for J = 1, N
    A[I,J] = A[I-1,J+1] + C
```

Loop interchange is illegal since $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix} = \begin{bmatrix}-1\\1\end{bmatrix}$

Let us try loop interchange followed by loop reversal. The transformation matrix T is $\begin{bmatrix}-1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} = \begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}$.

```
for J = N, 1
  for I = 1, N
    A[I,J] = A[I-1,J+1] + C
```

Applying T to the loop nest is legal since $\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix}$.

Challenges for Real-World Compilers

- Conditional execution
- Symbolic loop bounds
- Indirect memory accesses
- …

References

- R. Allen and K. Kennedy – Optimizing Compilers for Multicore Architectures, Chapters 5-6.
- S. Midkiff – Automatic Parallelization: An Overview of Fundamental Compiler Techniques.
- P. Sadayappan and A. Sukumaran Rajam – CS 5441: Parallel Computing, Ohio State University.
- M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. TPDS'91.