Data Dependences

- Essential constraints:

S1: \( a = b + c \)
S2: \( d = a \times 2 \)
S3: \( a = c + 2 \)
S4: \( e = d + c + 2 \)
Data Dependences

- Essential constraints:
  
  S1: \( a = b + c \)
  S2: \( d = a \times 2 \)
  S3: \( a = c + 2 \)
  S4: \( e = d + c + 2 \)

- S2 must execute after S1
Data Dependences

- Essential constraints:

S1:  \( a = b + c \)
S2:  \( d = a \times 2 \)
S3:  \( a = c + 2 \)
S4:  \( e = d + c + 2 \)

- S3 must execute after S2
Data Dependences

- Essential constraints:

  S1: \( a = b + c \)
  S2: \( d = a \times 2 \)
  S3: \( a = c + 2 \)
  S4: \( e = d + c + 2 \)

- S3 must execute after S1
Data Dependences

• Essential constraints:

\[ S1: \quad a = b + c \]
\[ S2: \quad d = a \times 2 \]
\[ S3: \quad a = c + 2 \]
\[ S4: \quad e = d + c + 2 \]

• But S3 and S4 can execute in either order, or concurrently
Data Dependences

- Essential constraints:

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \times 2 \\
S3: & \quad a = c + 2 \\
S4: & \quad e = d + c + 2
\end{align*}
\]

- S1 and S2 cannot execute concurrently
- S2 and S3 cannot execute concurrently
- S1 and S3 cannot execute concurrently
- But S3 and S4 can execute concurrently
- Execution conditions due to Bernstein (1966)
Types of Dependences

- **Flow-dependence** occurs when a variable which is assigned a value in one statement say S1 is read in another statement, say S2 later.

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad d = a \times 3
\end{align*}
\]
Types of Dependences

- **Anti-dependence** occurs when a variable which is read in one statement say S1 is assigned a value in another statement, say S2, later.

\[
\begin{align*}
S1: & \quad d = a \times 3 \\
S2: & \quad a = b + c
\end{align*}
\]
Types of Dependences

- **Output-dependence** occurs when a variable which is assigned a value in one statement say S1 is later reassigned in another statement, say S2.

\[
\begin{align*}
S1: & \quad a = b + c \\
S2: & \quad a = d \times 3
\end{align*}
\]
Types of Dependences

- **Input-dependence** occurs when a variable is read in two different statements say S1 and S2. **Relative ordering** of S1 and S2 is **not important** for input dependence.

\[
\begin{align*}
\text{S1:} & \quad a = b + c \\
\text{S2:} & \quad d = b \times 3
\end{align*}
\]
Data Dependences in Loops

- Associate a dynamic instance to each statement. For example

```
For i = 1 to 50
S1:   A(i) = B(i-1) + C(i)
S2:   B(i) = A(i+2) + C(i)
EndFor
```

- Statements S1 and S2 are executed 50 times. We say S2(10) to mean the execution of S2 when i = 10.
- Dependences are based on dynamic instances of statements.
Data Dependences in Loops

- Unrolling loops can help one figure out dependences:

\[
\begin{align*}
S1(1): & \quad A(1) = B(0) + C(1) \\
S2(1): & \quad B(1) = A(3) + C(1) \\
S1(2): & \quad A(2) = B(1) + C(2) \\
S2(2): & \quad B(2) = A(4) + C(2) \\
S1(3): & \quad A(3) = B(2) + C(3) \\
S2(3): & \quad B(3) = A(5) + C(3) \\
& \quad \vdots \\
S1(50): & \quad A(50) = B(49) + C(50) \\
S2(50): & \quad B(50) = A(52) + C(50)
\end{align*}
\]
Iteration Spaces

- Nested loops define an iteration space:
  ```plaintext
  For i = 1 to 4
    for j = 1 to 4
      A(i,j) = A(i,j) + C(j)
    Endfor
  Endfor
  ```

- Sequential execution (traversal order):
- Dimensionality of iteration space = loop nest level; arbitrary convex shapes are allowed
- Change in order of execution is valid if no dependences are violated
Single Processor Performance Enhancement

- Two fundamental issues:
  - Adequate fine-grained parallelism
    - Exploit vector instructions sets (SSE, AVX, AVX-512, ...)
    - Multiple pipelined functional units in each core
  - Minimize memory-access costs (about an order of magnitude higher than clock cycle)

- Useful loop transformations:
  - Loop Permutation
  - Loop Unrolling
  - Loop Blocking (tiling)
  - Loop Fusion/Distribution
Access Stride and Spatial Locality

• Access stride: Separation between successively accessed memory locations
• Unit access stride maximizes spatial locality (only one miss per cache line)
• 2-D arrays have different linearized representations in Fortran and C

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
</tbody>
</table>

Row Major Order (C)

| a | b | c | d | e | f | o | p |

| a | e | i | m | b | f | l | p |

Column-major Order (Fortran)
Matrix-Vector Multiplication: Dot-Product

For \( I = 1, N \)
   For \( J = 1, N \)
      \( y(I) = y(I) + A(I,J) \times x(J) \)
   EndFor
EndFor

Access Stride for Arrays

\[
\begin{array}{c|ccc}
   & A & x & y \\
\hline
  C & 1 & 1 & 0 \\
Fortran & n & 1 & 0 \\
\end{array}
\]
Matrix-Vector Multiplication: SAXPY

For J = 1, N
    For I = 1, N
        y(I) = y(I) + A(I,J) * x(J)
    EndFor
EndFor

Access Stride for Arrays

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>n</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Fortran</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Loop Permutation: Matrix Multiplication

for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        for(k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];

<table>
<thead>
<tr>
<th>Reference</th>
<th>ikj</th>
<th>kij</th>
<th>jik</th>
<th>ijk</th>
<th>jki</th>
<th>kji</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(i,j)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>A(i,k)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>B(k,j)</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>n</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Access Stride for Arrays (C: Row-Major)
## Loop Permutation: Matrix Multiplication

<table>
<thead>
<tr>
<th>Reference</th>
<th>ikj</th>
<th>kij</th>
<th>jik</th>
<th>ijk</th>
<th>jki</th>
<th>kji</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(i,j)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>A(i,k)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>B(k,j)</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td>n</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Best**

**Worst**

**Access Stride for Arrays (C: Row-Major)**

<table>
<thead>
<tr>
<th>Compiler/Opt</th>
<th>ikj</th>
<th>kij</th>
<th>jik</th>
<th>ijk</th>
<th>jki</th>
<th>kji</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc -fast</td>
<td>17.0</td>
<td>17.0</td>
<td>17.0</td>
<td>17.0</td>
<td>17.0</td>
<td>17.0</td>
</tr>
<tr>
<td>icc -O3</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>icc -O2</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
</tr>
<tr>
<td>icc -O1</td>
<td>2.0</td>
<td>2.0</td>
<td>.95</td>
<td>1.0</td>
<td>.29</td>
<td>.29</td>
</tr>
<tr>
<td>gcc -O3</td>
<td>6.1</td>
<td>7.6</td>
<td>.94</td>
<td>1.0</td>
<td>.29</td>
<td>.29</td>
</tr>
<tr>
<td>gcc -O2</td>
<td>2.0</td>
<td>2.0</td>
<td>.94</td>
<td>1.0</td>
<td>.29</td>
<td>.29</td>
</tr>
<tr>
<td>gcc -O1</td>
<td>1.9</td>
<td>1.9</td>
<td>.94</td>
<td>1.0</td>
<td>.29</td>
<td>.29</td>
</tr>
</tbody>
</table>

Performance on one core of Intel Xeon x5650 (GFLOPS)
Permutation: Non-Rectangular Loops

\[
\begin{align*}
\text{For } & I = 1, N \\
& \text{For } J = 1, I \\
& \quad y(I) = y(I) + A(I,J) \times x(J) \\
& \text{EndFor} \\
& \text{EndFor}
\end{align*}
\]
Permutation: Non-Rectangular Loops

For I = 1, N
   For J = 1, I
      y(I) = y(I) + A(I,J) * x(J)
   EndFor
EndFor

For J = 1, N
   For I = J, N
      y(I) = y(I) + A(I,J) * x(J)
   EndFor
EndFor
Transformations: Loop Fusion

- Fusion: Fuses two loops, also known as jamming (useful for locality enhancement). In example below, after fusion, you cannot have dependencies from S2 to S1

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

```
For I = 1, N
  a = B(I)+C(I)
  E(I) = a*D(I)
  EndFor
```
Illegal Loop Fusion Example

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

We have flow dependences from S1 to S2
Illegal Loop Fusion Example

For $I = 1, N$

S1: $A(I) = B(I) + C(I)$
EndFor

For $I = 1, N$

S2: $E(I) = A(I+1) \cdot D(I)$
EndFor

*Illegal fusion*: On fusing the two loops, we have a violation of original data dependence

We have flow dependences from S1 to S2
Transformations: Loop Distribution

- **Loop Distribution:** Splits a single loop nest into many, also known as *loop fission.*

  \[
  \text{For } I = 1, N \\
  \text{S1: } A(I) = B(I) + C(I) \\
  \text{S2: } E(I) = A(I) \times D(I) \\
  \text{EndFor}
  \]

- Like loop fusion, distribution is not always legal – must ensure that no data dependences are violated.

- **Needed for vectorization**
Loop Unrolling

• Reduce number of iterations of loop but add statement(s) to loop body to do work of missing iterations
• Increases amount of instruction-level parallelism in loop body

```c
for(j=0; j< 2*m; j++)
{
    Loop-Body(j)
}
```

```c
for(j=0; j< 2*m; j+=2)
{
    Loop-Body(j)
    Loop-Body(j+1)
}
```

```c
for(i=0; i< n; i++)
    for(j=0; j< 2*m; j++)
    {
        Loop-Body(i,j)
    }
```

```c
for(i=0; i< n; i++)
    for(j=0; j< 2*m; j+=2)
    {
        Loop-Body(i,j)
        Loop-Body(i,j+1)
    }
```
Example: Inner Loop Unrolling

```c
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];
        y[i]=y[i]+a[i][j+1]*x[j+1];
        y[i]=y[i]+a[i][j+2]*x[j+2];
        y[i]=y[i]+a[i][j+3]*x[j+3]; }
```

// Assumes n is a multiple of 4
Outer Loop Unrolling (Unroll/Jam)

- Reduce number of iterations of an outer loop
- Simply replicating inner-loop structures will not increase op-level parallelism; need to fuse ("jam") replicated inner-loops
- Changes memory access order
  - Could reduce cache misses
  - Hence must verify validity of transformation

```plaintext
for(i=0; i<2*n; i+=2)  
{for(j=0; j<m; j++)
   Loop-Body(i,j)    // 2-way outer-unroll
   for(j=0; j<m; j++)  // does not increase
       Loop-Body(i+1,j) // op-lvl parallelism
}
```

```plaintext
for(i=0; i<2*n; i+=2)  
{for(j=0; j<m; j++)
   {Loop-Body(i,j)    // unroll-jam increases
   Loop-Body(i+1,j)} // op-lvl parallelism
}
```
Example: Outer Loop Unrolling

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

// Assumes n is a multiple of 4
for(i=0; i<n; i+=4)
    for(j=0; j<n; j++) {
        y[i]=y[i]+a[i][j]*x[j];
        y[i+1]=y[i+1]+a[i+1][j]*x[j];
        y[i+2]=y[i+2]+a[i+2][j]*x[j];
        y[i+3]=y[i+3]+a[i+3][j]*x[j];
    }
Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- “block” (in this context) does not mean “cache block”.
- Instead, it means a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
=
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Key idea: Sub-blocks (i.e., \textit{Axy}) can be treated just like scalars.

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} & C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} & C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]
Blocked Matrix Multiplication

/* ijk */
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];

/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
    for (jt=0; jt<n; jt+=T)
        for (kt=0; kt<n; kt+=T)
            for (i=it; i<it+T; i++)
                for (j=jt; j<jt+T; j++)
                    for (k=kt; k<kt+T; k++)
                        c[i][j] += a[i][k]*b[k][j];
Cache Misses: Blocked Mat-Mult

/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
  for (jt=0; jt<n; jt+=T)
    for (kt=0; kt<n; kt+=T)
      for (i=it; i<it+T; i++)
        for (j=jt; j<jt+T; j++)
          for (k=kt; k<kt+T; k++)
            c[i][j] += a[i][k]*b[k][j];

Assume fully associative Cache of size > 3*T*T

• Each sub-mat-mult involves product of two TxD sub-matrices of A,B to contribute to a TxD sub-matrix of C
• Each sub-mat-mult has at most 3*(T2/B) cache misses (no evictions during computation; T2 elements for each array)
• Number of result blocks of C: (N/T)*(N/T) = N^2/T^2
• Each C-block requires (N/T) sub-mat-mults
• Total cache misses \( \leq 3*(T2/B)*(N/T)*N^2/T^2 = 3N^3/(B*T) \)
• T can be as large as sqrt(CacheSize/3)
Tiling = Loop-Split+Permutation

/* ijk */
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];

for (it=0; it<n; it+=T)
  for (jt=0; jt<n; jt+=T)
    for (kt=0; kt<n; kt+=T)
      for (i=it; i<it+T; i++)
        for (j=jt; j<jt+T; j++)
          for (k=kt; k<kt+T; k++)
            c[i][j]+= a[i][k]*b[k][j];

for (it=0; it<n; it+=T)
  for (jt=0; jt<n; jt+=T)
    for (kt=0; kt<n; kt+=T)
      for (i=it; i<it+T; i++)
        for (j=jt; j<jt+T; j++)
          for (k=kt; k<kt+T; k++)
            c[i][j]+= a[i][k]*b[k][j];

Strip-mine each loop into a pair of equivalent loops

for (it=0; it<n; it+=T)
  for (i=it; i<it+T; i++)
    for (jt=0; jt<n; jt+=T)
      for (j=jt; j<jt+T; j++)
        for (kt=0; kt<n; kt+=T)
          for (k=kt; k<kt+T; k++)
            c[i][j]+= a[i][k]*b[k][j];

Loop Permutation
Total Cache Miss Analysis: IJK

let: \( C < B \times N \)

fully associative cache

I
J
K

for (i = 0; j < N; i++)
for (j = 0; j < N; j++)
for (k = 0; k < N; k++)
\( C[i][j] += A[i][k] \times B[k][j] \)

k loop:

I
J
K

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>K</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

N2
N3
N2

B
B
B
Total Cache Miss Analysis: JKI

Let: \( C < B^*N \)
fully associative cache

\[
\begin{align*}
\text{for } (j = 0; j < N; j++) & \\
\quad \text{for } (k = 0; k < N; k++) & \\
\quad \quad \text{for } (i = 0; i < N; i++) & \\
\quad \quad \quad C[i][j] & += A[i][k] \times B[k][j]
\end{align*}
\]
/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
    for (jt=0; jt<n; jt+=T)
        for (kt=0; kt<n; kt+=T)
            for (i=it; i<it+T; i++)
                for (j=jt; j<jt+T; j++)
                    for (k=kt; k<kt+T; kt++)
                        c[i][j] += a[i][k]*b[k][j];
Blocked Matrix Multiply: Cache Misses

/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
    for (jt=0; jt<n; jt+=T)
        for (kt=0; kt<n; kt+=T)
            for (i=it; i<it+T; i++)
                for (j=jt; j<jt+T; j++)
                    for (k=kt; k<kt+T; kt++)
                        c[i][j]+= a[i][k]*b[k][j];

Assume fully associative Cache: size > 3*T*T
But size < T*N

<table>
<thead>
<tr>
<th>Loop</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>N/T</td>
<td>N/T</td>
<td>N/T</td>
</tr>
<tr>
<td>jt</td>
<td>N/T</td>
<td>N/T</td>
<td>N/T</td>
</tr>
<tr>
<td>kt</td>
<td>N/T</td>
<td>N/T</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>T</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>T/B</td>
<td>T/B</td>
</tr>
<tr>
<td>k</td>
<td>T/B</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>N^3/(TB)</td>
<td>N^3/(TB)</td>
<td>N^2/(TB)</td>
</tr>
</tbody>
</table>
Tiling: Arbitrary Bounds and Tilesize

/* ijk */
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        for (k=0; k<p; k++)
            c[i][j] += a[i][k]*b[k][j];

for (it=0; it<n; it+=Ti)
    for (jt=0; jt<m; jt+=Tj)
        for (kt=0; kt<p; kt+=Tk)
            for (i=it; i< min(it+Ti,n); i++)
                for (j=jt; j< min(jt+Tj,m); j++)
                    for (k=kt; k< min(kt+Tk,p); k++)
                        c[i][j] += a[i][k]*b[k][j];