Lecture Overview

• Very Brief Introduction to Dependences
• Loop Transformations
The Big Picture

• What are our goals?
  • Simple goals: Make execution time as small as possible

• Which leads to:
  • Achieve execution of many (all, in the best case) instructions in parallel
  • Find independent instructions
Dependences

• We will concentrate on data dependences
• Simple example of data dependence:
  \[ S_1: \pi = 3.14; \]
  \[ S_2: r = 5.0; \]
  \[ S_3: \text{area} = \pi \times r \times r; \]

• Statement \( S_3 \) cannot be moved before either \( S_1 \) or \( S_2 \) without compromising correct results
Dependences

Formally:

• There is a **data dependence** from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if:
  1. Both statements access the same memory location and at least one of them writes into it, and
  2. There is a feasible run-time execution path from $S_1$ to $S_2$
Load Store Classification

- Dependences classified in terms of load-store order:
  - True dependence ("read after write" = RAW hazard)
  - Antidependence (WAR hazard)
  - Output dependence (WAW hazard)
Examples

• True dependence
  \[ a = 3 \]
  \[ b = a \times 2 \]

• Anti dependence
  \[ a = b \times 3 \]
  \[ b = 8 \]

• Output dependence
  \[ a = b \times c \]
  \[ a = e + f \]
Dependence in Loops

• Consider this loop:

```c
for (i = 0; i < N-1; i++)
    A[i+1] = A[i] + B[i]; // S_1
```

• Iteration \( i+1 \) reads the element of \( A \) value written by iteration \( i \)

• \( S_1 \) is a **loop-carried dependence**
Transformations

- Loops (and other instructions) can be transformed as long as order of dependences does not change
Reordering Transformations

- Any program transformation that changes the order of execution of the code, without adding or removing any executions of any statements

- Does not eliminate dependences
- But, it can change the ordering of the dependence, which will lead to incorrect behavior

- Preserves a dependence if it preserves the relative execution order of the source and sink of that dependence
Loop Transformations

• Compilers have always focused on loops
  • Higher execution counts
  • Repeated, related operations
• Much of real work takes place in loops
Several effects to attack

- Overhead
  - Decrease control-structure cost per iteration
- Locality
  - Spatial locality ⇒ use of co-resident data
  - Temporal locality ⇒ reuse of same data
- Parallelism
  - Execute independent iterations of loop in parallel
Eliminating Overhead

• Loop unrolling
• To reduce overhead, replicate the loop body

for (i = 0; i < 100; i++) {
}

for (i = 0; i < 100; i+=4) {
  A[i+1] = A[i+1] + B[i+1];
}

• Sources of Improvement
• Less overhead per useful operation
• Longer basic blocks for local optimization
Eliminating Overhead

- Loop unrolling with unknown bounds
- Generate guard loops

```c
for (i = 0; i+4 < n; i += 4) {
    A[i+1] = A[i+1] + B[i+1];
}
for (; i < n; i++) {
}
```
Loop Unswitching

- Hoist invariant control-flow out of loop nest
- Replicate the loop & specialize it
- No tests, branches in loop body
- Longer segments of straight-line code
Loop Unswitching

```c
if (c)
   for (...) {
      S0;
      S1;
   } else
   for (...) {
      S0;
      S2;
   }
else
   for (...) {
      S1;
      S3;
   }
```
Loop Unswitching

```c
for (i = 0; i < N; i++) {
    if (c) D[i] = 0;
}
```

if (c) {
    for (i = 0; i < N; i++) {
        D[i] = 0;
    }
} else {
    for (i = 0; i < N; i++) {
    }
}
```
Loop fusion

- Two loops over same iteration space ⇒ one loop
- Safe if does not change the values used or defined by any statement in either loop (i.e., does not violate dependences)

```c
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
for (j = 0; j < N; i++) {
    D[j] = A[j] * C[j];
}
```

For big arrays, $A[j]$ may not be in the cache

$A[i]$ will be in the cache
Loop Fusion Advantages

• Enhance temporal locality
• Reduce control overhead
• Longer blocks for local optimization & scheduling
• Can convert inter-loop reuse to intra-loop reuse
Loop distribution (fission)

- Single loop with independent statements $\Rightarrow$ multiple loops
- Starts by constructing statement level dependence graph
- Safe to perform distribution if:
  - No cycles in the dependence graph
  - Statements forming cycle in dependence graph put in same loop
Loop distribution (fission)

for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
    F[i] = D[i] * E[i];
}

for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}

for (j = 0; j < N; i++) {
    F[j] = D[j] * E[j];
}
Loop distribution (fission)

(1) for (i = 1; i < N; i++) {
(3) \[ B[i] = C[i-1] \times X + C \]
(4) \[ C[i] = \frac{1}{B[i]} \]
(5) \[ D[i] = \sqrt{C[i]} \]
(6) }

Has the following dependence graph
Loop distribution (fission)

(1) for (i = 1; i < N; i++) {
(3)    B[i] = C[i-1]*X+C
(4)    C[i] = 1/B[i]
(5)    D[i] = sqrt(C[i])
(6) }

for (i = 1; i < N; i++) {
}
for (i = 1; i < N; i++) {
    B[i] = C[i-1]*X+C
    C[i] = 1/B[i]
}
for (i = 1; i < N; i++) {
    D[i] = sqrt(C[i])
}
Loop Fission Advantages

• Enables other transformations
  • e.g., vectorization

• Resulting loops have smaller cache footprints
  • More reuse hits in the cache
Loop Interchange

for (i = 1; i < 50; i++)
    for (j = 0; j < 100; j++)
        A[i][j] = B[i][j] * C[i][j];

Swap inner & outer loops to rearrange iteration space

for (j = 0; j < 100; j++)
    for (i = 1; i < 50; i++)
        A[i][j] = B[i][j] * C[i][j];
Loop Interchange Effect

• If one loop carries all dependence relations
  • Swap to outermost loop and all inner loops executed in parallel
• If outer loops iterates many times and inner only a few
  • Swap outer and inner loops to reduce startup overhead
• Improves reuse by using more elements per cache line
• Goal is to get as much reuse into inner loop as possible
Reordering Loops for Locality

Consider an array in column-major order (Fortran):

```plaintext
for (i ...)
  for (j ...)
    A[i][j]
```

As little as 1 used element per line

```plaintext
for (j ...)
  for (i ...)
    A[i][j]
```

Runs down cache line

In **row-major order** (C), the opposite loop ordering causes the same effects
Loop permutation

- Interchange is degenerate case
  - Two perfectly nested loops
- More general problem is called permutation
- Safety
  - Permutation is safe iff no data dependences are reversed
  - Preserves flow of data from definitions to uses
Loop Permutation Effects

- Change order of access & order of computation
- Move accesses closer in time ⇒ increase temporal locality
- Move computations farther apart ⇒ cover pipeline latencies
Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row is in contiguous memory locations
- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    sum += a[0][i];
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    sum += a[i][0];
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kij)

/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

Inner loop:

(i,k)  (k,*)  (i,*)
A       B       C

Fixed   Row-wise  Row-wise

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ikj)

/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = **1.25**

```c
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = **0.5**

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

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = **2.0**

```c
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```
Pentium Matrix Multiply Performance

Cycles/iteration vs. Array size (n)

- kji
- jki
- kij
- ikj
- jik
- jik
- ijk

Better
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., \( A_{xy} \)) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Strip Mining

\[
\begin{align*}
    &\text{for } (j = 0; j < 100; j++) \\
    &\quad \text{for } (i = 1; i < 50; i++) \\
    &\quad \quad A[i][j] = B[i][j] * C[i][j]; \\
    \\
    &\text{for } (j = 0; j < 100; j++) \\
    &\quad \text{for } (ii = 0; ii < 50; ii += 8) \\
    &\quad \quad \text{for } (i = ii; i < \min(ii+8,50); i++) \\
    &\quad \quad \quad A[i][j] = B[i][j] * C[i][j]; \\
\end{align*}
\]

Blocking = strip mining + interchange
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++)
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
            c[i][j] += sum;
    }
}
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a $1 \times \text{bsize}$ sliver of $A$ by a $\text{bsize} \times \text{bsize}$ block of $B$ and accumulates into $1 \times \text{bsize}$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```plaintext
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bsize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
}
```

- For (i=0; i<n; i++) {
  - for (j=jj; j < min(jj+bsize,n); j++) {
    - sum = 0.0
    - for (k=kk; k < min(kk+bsize,n); k++) {
      - sum += a[i][k] * b[k][j];
    }
    - c[i][j] += sum;
  }
}

- **Innermost Loop Pair**
  - **Tuesday, May 4, 2010**
Pentium Blocked Matrix Multiply Performance

- Blocking improves performance by a factor of two
- relatively insensitive to array size

**Diagram:**
- **Graph:**
  - X-axis: Array size (n)
  - Y-axis: Cycles/iteration

**Legend:**
- kji
- jki
- kij
- ikj
- jik
- ijk
- bijk (bsize = 25)
- bikj (bsize = 25)
Observations

- Compiler (and programmer) can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)