Advanced Topics

Optimization for parallel machines and memory hierarchies

Last Time

• Dependence analysis

Today

• Loop transformations

• An example - McKinley, Carr, Tseng loop transformations to improve cache performance

Which Loops are Parallel?

<table>
<thead>
<tr>
<th>do I = 1, N</th>
</tr>
</thead>
<tbody>
<tr>
<td>do J = 1, N</td>
</tr>
</tbody>
</table>

S1  \[ A(I,J) = A(I,J-1) + 1 \]

S2  \[ A(I,J) = A(I-1,J-1) + 1 \]

S3  \[ B(I,J) = B(I-1,J+1) + 1 \]

• A dependence \( D = (d_1, \ldots, d_k) \) is carried at level \( i \), if \( d_i \) is the first nonzero element of the distance/direction vector.

• A loop \( l_i \) is parallel, if \( \not\exists \) a dependence \( D_j \) carried at level \( i \). Either

\[
\begin{array}{c|c|c}
\forall D_j & \text{distance vector} & \text{direction vector} \\
\hline
\geq 0 & d_1, \ldots, d_{i-1} & d_1, \ldots, d_{i-1} = "<" \\
\text{OR} & d_1, \ldots, d_i & d_1, \ldots, d_i = "=" \\
\end{array}
\]
Loop Transformations

**Taxonomy**
- Loop unrolling
- Loop interchange
- Loop fusion
- Loop distribution (a.k.a. fission)
- Loop skewing
- Strip mine and interchange (a.k.a. tiling & blocking)
- Unroll-and-jam (a variety of tiling)
- Loop reversal

**Loop Interchange**

Do $i = 1, N$
Do $j = 1, N$

$S_1 A(i, j) = A(i-1, j) + 1$
Enddo
Enddo

$S_2 B(i, j) = B(i-1, j+1) + 1$
Enddo
Enddo

Loop interchange is safe *iff*

- it does not reverse the execution order of the source and sink of any dependence in the nest, i.e., if the distance vector would become negative.
  - Enables parallelization of outer and/or inner loops
  - Changes execution order of the statements
  - Can improve reuse
Loop Fusion

\[
\begin{align*}
\text{do } i &= 2, n \\
s_1 & \quad a(i) = b(i) \\
\text{do } i &= 2, n \\
s_2 & \quad c(i) = b(i) \ast a(i-1) \\
\end{align*}
\]

\[\Rightarrow \text{ loop fusion } \Rightarrow \]

\[
\begin{align*}
\text{do } i &= 2, n \\
s_1 & \quad a(i) = b(i) \\
\text{do } i &= 2, n \\
s_2 & \quad c(i) = b(i) \ast a(i-1) \\
\end{align*}
\]

\[\Leftarrow \text{ loop distribution } \Leftarrow \]

**Loop Fusion** is safe iff

- no forward dependence between nests becomes a backward loop carried dependence.

\[\Rightarrow \text{ Would fusion be safe if } s_2 \text{ referenced } a(i+1) ? \]

- Benefits
  - Reuse
  - Eliminates synchronization between parallel loops
  - Reduced loop overhead

Loop Distribution

\[
\begin{align*}
\text{do } i &= 2, n \\
s_1 & \quad a(i) = b(i) \\
s_2 & \quad c(i) = b(i) \ast a(i+1) \\
\text{do } i &= 2, n \\
s_2 & \quad c(i) = b(i) \ast a(i+1) \\
\text{do } i &= 2, n \\
s_1 & \quad a(i) = b(i) \\
\end{align*}
\]

\[\Rightarrow \text{ loop distribution } \Rightarrow \]

**Loop Distribution** is safe iff

- statements involved in a cycle of dependences (*recurrence*) remain in the same loop, &

- if \[\exists\] a dependence between two statements placed in different loops, it must be forward.

- Benefits
  - Partial parallelization
  - Reduces resource requirements
  - Enables other transformations
Loop Skewing and Interchange

\[
\begin{align*}
\text{do } I & = 1, 100 \\
\text{do } J & = 2, 100 \\
A(I,J) & = A(I-1,J) + A(I,J-1)
\end{align*}
\]

Loop skewing is always safe.

- The original dependences, \((1,0) \& (0,1)\) prevent interchange and parallelization.
- It changes the dependences to \((1,1) \& (0,1)\) and after interchange, \((1,1) \& (1,0)\), the inner loops is parallel.
  \(\Rightarrow\) Enables inner loop parallelization

Strip Mine and Interchange

\[
\begin{align*}
\text{do } I & = 1, n \\
\text{do } J & = 1, n \\
A(J,I) & = B(J) \times C(I)
\end{align*}
\]

\[
\begin{align*}
\text{do } II & = 1, n, \text{ tile} \\
\text{do } I & = II, II + \text{tile} -1 \\
\text{do } J & = 1, n \\
A(J,I) & = B(J) \times C(I)
\end{align*}
\]

Strip Mining is always safe. With interchange it

- enables loop invariant reuse by changing the shape of the iteration space
**Improving Reuse with Loop Transformations**

**Motivation:** Enable portable programming without sacrificing performance

- optimization framework
- cache model
- compound loop transformation algorithm
  - permutation
  - fusion
  - distribution
  - reversal
- results
  - transformation
  - simulation
  - performance

**Optimization Framework**

1. improve order of memory accesses to exploit all levels of the memory hierarchy
   \[\implies \text{cache line size}\]

2. Tile to fit in cache, second level cache, TLB
   \[\implies \text{size of cache(s), replacement policy, associativity,}\]

3. register tiling via unroll-and-jam and scalar replacement
   \[\implies \text{number and type of registers}\]

**Assumptions**

- cls - the cache line size in terms of data items
- Fortran column-major order
- interference occurs rarely for small numbers of inner loop iterations
Reuse

- temporal locality  
  *reuse of a specific location*
- spatial locality  
  *reuse of adjacent locations* (cache lines)

\[
do j = 1, 100 \\
  do k = 1, 100 \\
    do i = 1, 100 \\
      c(i,j) = c(i,j) + a(i,k) \times b(k,j)
\]

To Determine Temporal and Spatial Reuse:

for each loop \( l \) in a nest, consider \( l \) innermost

- group references with temporal locality  
  \( \Rightarrow \) reference groups
- compute the cost in cache lines accessed  
  \( \Rightarrow \) loop cost
- rank the loops based on their cost  
  \( \Rightarrow \) memory order

Loop Cost of \text{dmxpy From Linpackd}

Loop Cost in cache lines, \( \text{cls} = 4 \)

\[
do j = 1, n2 \\
  do i = 1, n1 \\
    y(i) = y(i) + x(j) \times m(i,j)
\]

<table>
<thead>
<tr>
<th>reference group</th>
<th>loop i</th>
<th>loop j</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y(i) )</td>
<td>1/4 ( n1 \times n2 )</td>
<td>1 \times n1</td>
</tr>
<tr>
<td>( x(j) )</td>
<td>1 \times n2</td>
<td>1/4 ( n2 \times n1 )</td>
</tr>
<tr>
<td>( m(i,j) )</td>
<td>1/4 ( n1 \times n2 )</td>
<td>n2 \times n1</td>
</tr>
</tbody>
</table>

**total loop cost**  
\( 1/2 \times n1 \times n2 + n2 \times n1 + 5/4 \times n1 \times n2 + n1 \)
Loop permutation

**key insight:** If loop $l$ promotes more reuse than loop $k$ at the innermost position, then it will also promote more reuse at any outer position.

- memory order
  - (a) is it legal?
  - (b) if not find a nearbyPermutation $O(n^2)$

$\Rightarrow$ avoids combinatorial search present in other algorithms

NearbyPermutation

**Input:**
- $O = \{i_1, i_2, ..., i_n\}$, the original loop ordering
- $DV = \text{set of original legal direction vectors for } l_o$
- $L = \{i_{\sigma_1}, i_{\sigma_2}, ..., i_{\sigma_n}\}$, a permutation of $O$

**Output:**
- $P$ a nearby permutation of $O$

**Algorithm:**
- $P = \emptyset$ ; $k = 0$ ; $m = n$
- while $L \neq \emptyset$
  - for $j = 1..m$
    - $l = l_j \in L$
    - if direction vectors for $\{p_1, ..., p_k, l\}$ are legal
      - $P = \{p_1, ..., p_k, l\}$
      - $L = L - \{l\}$ ; $k = k + 1$ ; $m = m - 1$
      - break for
  - endif
- endfor
- endwhile
Matrix Multiply - execution times in seconds

**Execution Times** (in seconds) vs. Loop Organization

Sun Sparc2
Iter i860
IBM RS/6000

JKI JKI JIK IJK KIJ IKJ - ORDER

CS 380C Lecture 24
15
Locality Analysis

Loop Fusion

Fortran 90 loops for ADI Integration

DO I = 2, N
X(I,1:N) = X(I,1:N) - X(I-1,1:N)*A(I,1:N)/B(I-1,1:N)

DO I = 2, N ↓ *simple translation to Fortran 77*
DO K = 1, N
X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
DO K = 1, N
B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)

DO K = 1, N ↓ *loop fusion & interchange*
DO I = 2, N
X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)

Execution times in seconds.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Original</th>
<th>Distributed</th>
<th>Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun Sparc2</td>
<td>.806</td>
<td>.813</td>
<td>.672</td>
</tr>
<tr>
<td>Intel i860</td>
<td>.547</td>
<td>.548</td>
<td>.518</td>
</tr>
<tr>
<td>IBM RS6000</td>
<td>.390</td>
<td>.400</td>
<td>.383</td>
</tr>
</tbody>
</table>

CS 380C Lecture 24
16
Locality Analysis
**Loop Distribution**

\[
\text{DO } K = 1, N \\
A(K,K) = \sqrt{A(K,K)} \\
\text{DO } I = K+1, N \\
A(I,K) = A(I,K)/A(K,K) \\
\text{DO } J = K+1, I \\
A(I,J) \leftarrow A(I,K)*A(J,K) \\
\]

**Cholesky Factorization**

\[
\text{DO } K = 1, N \\
A(K,K) = \sqrt{A(K,K)} \\
\text{DO } I = K+1, N \\
A(I,K) = A(I,K)/A(K,K) \\
\text{DO } J = K, N \\
\text{DO } I=J+1, N \\
A(I,J) \leftarrow A(I,K)*A(J,K) \\
\]

⇒ loop distribution & triangular interchange

⇒ Execution times (in seconds)

<table>
<thead>
<tr>
<th>Algorithm Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each nest $L_i$ in a set of adjacent nests</td>
</tr>
<tr>
<td>• compute reference groups for each $l_i$</td>
</tr>
<tr>
<td>• compute loop cost for each $l_i$ and sort</td>
</tr>
<tr>
<td>• permutation with reversal?</td>
</tr>
<tr>
<td>• fuse inner loops and permute?</td>
</tr>
<tr>
<td>• distribute and permute?</td>
</tr>
<tr>
<td>fuse nests $L_i$?</td>
</tr>
</tbody>
</table>

Execution times (in seconds)

Sun Sparc2 Intel i860 IBM RS/6000
KJI JKI KJ IKJ JIK IJK - ORDER
Results

test suite

- Perfect Benchmarks
- SPEC Benchmarks
- NAS Benchmarks
- 4 additional programs

experiments

- on our ability to transform programs
- simulated hit rates for RS/6000 and i860
- execution times on an RS/6000

Achieving Memory Order for Loop Nests

<table>
<thead>
<tr>
<th>Percentage of Loop Nests in Memory Order</th>
<th>Original</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 20</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>&gt;= 40</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>&gt;= 60</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>&gt;= 70</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>&gt;= 80</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>&gt;= 90</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>
Achieving Memory Order for Inner Loops

Simulated Cache Hit Rates

cache1: RS/6000 64K cache, 4-way, 128 byte cache line
cache2: i860 8K cache, 2-way, 32 byte cache line

for RS/6000

⇒ in 12 of 27 programs, optimized procedures started with 100% hit rates

⇒ 98.86 – average hit rate for original programs

<table>
<thead>
<tr>
<th>Perfect Club</th>
<th>Memory Order</th>
<th>Org</th>
<th>Prm</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>adm</td>
<td>Perfect Club</td>
<td>52</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>arc2d</td>
<td></td>
<td>55</td>
<td>28</td>
<td>17</td>
</tr>
<tr>
<td>bdna</td>
<td></td>
<td>75</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>dyfesm</td>
<td></td>
<td>63</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>flo52</td>
<td></td>
<td>83</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>mdg</td>
<td></td>
<td>83</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>mg3d</td>
<td></td>
<td>95</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ocean</td>
<td></td>
<td>82</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>qcd</td>
<td></td>
<td>53</td>
<td>11</td>
<td>36</td>
</tr>
<tr>
<td>spec77</td>
<td></td>
<td>64</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>track</td>
<td></td>
<td>50</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>trfd</td>
<td></td>
<td>52</td>
<td>0</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opt. Proc. Cache 1</th>
<th>Hit Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache 1</td>
</tr>
<tr>
<td></td>
<td>org</td>
</tr>
<tr>
<td>adm</td>
<td>100</td>
</tr>
<tr>
<td>arc2d</td>
<td>100</td>
</tr>
<tr>
<td>bdna</td>
<td>100</td>
</tr>
<tr>
<td>dyfesm</td>
<td>100</td>
</tr>
<tr>
<td>flo52</td>
<td>100</td>
</tr>
<tr>
<td>mdg</td>
<td>100</td>
</tr>
<tr>
<td>mg3d</td>
<td>100</td>
</tr>
<tr>
<td>ocean</td>
<td>100</td>
</tr>
<tr>
<td>qcd</td>
<td>100</td>
</tr>
<tr>
<td>spec77</td>
<td>100</td>
</tr>
<tr>
<td>track</td>
<td>100</td>
</tr>
<tr>
<td>trfd</td>
<td>99.9</td>
</tr>
</tbody>
</table>
### Performance Results in Seconds on RS6000

<table>
<thead>
<tr>
<th>Program</th>
<th>Original</th>
<th>Transformed</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc2d</td>
<td>410.13</td>
<td>190.69</td>
<td>2.15</td>
</tr>
<tr>
<td>dyfesm</td>
<td>25.42</td>
<td>25.37</td>
<td>1.00</td>
</tr>
<tr>
<td>flo52</td>
<td>62.06</td>
<td>61.62</td>
<td>1.01</td>
</tr>
<tr>
<td>dna7 (btrix)</td>
<td>36.18</td>
<td>30.27</td>
<td>1.20</td>
</tr>
<tr>
<td>dna7 (emit)</td>
<td>16.46</td>
<td>16.39</td>
<td>1.00</td>
</tr>
<tr>
<td>dna7 (gmtry)</td>
<td>155.30</td>
<td>17.89</td>
<td>8.68</td>
</tr>
<tr>
<td>dna7 (vpenta)</td>
<td>149.68</td>
<td>115.62</td>
<td>1.29</td>
</tr>
<tr>
<td>applu</td>
<td>146.61</td>
<td>149.49</td>
<td>0.98</td>
</tr>
<tr>
<td>appsp</td>
<td>361.43</td>
<td>337.84</td>
<td>1.07</td>
</tr>
<tr>
<td>linpackd</td>
<td>159.04</td>
<td>157.48</td>
<td>1.01</td>
</tr>
<tr>
<td>simple</td>
<td>963.20</td>
<td>850.18</td>
<td>1.13</td>
</tr>
<tr>
<td>wave</td>
<td>445.94</td>
<td>414.60</td>
<td>1.08</td>
</tr>
</tbody>
</table>

### Summary

#### Recap of Transformation Results
- 80% of nests were permuted into memory order
- 85% of inner loops were permuted into memory order
- Loop permutation is the most effective optimization
- 229 candidates for fusion, resulting in 80 nests
- 23 nests were distributed, resulting in 52 nests

#### Observations
- Many programs started out with high hit ratios
- Smaller cache sizes result in higher improvements in hit rates

⇒ Regardless of the original target architecture, compiler optimizations improve locality for uniprocessors.
Next Time

Compiling for an EDGE Architecture