Program Transformations

Michael O’Boyle

February, 2014
Course Structure

- L1 Introduction and Recap, L2 Course Work
- 5 lectures on high level restructuring for parallelism and memory
- Dependence Analysis
- **Program Transformations - loop and arrays**
- Automatic vectorisation, parallelisation
- Speculative Parallelisation
Lecture Overview

• Classification of program transformations - loop and array

• Role of dependence

• Loop restructuring - changing the number/type of loop

• Iteration reordering - reordering the iterations scanned.

• Array transformations - data layout transformation

• Simplified presentation. Large number of technicalities. Applicability. Worth.
References

• Loop Distribution with arbitrary control-flow McKinley and Kennedy Supercomputing 1990


• A Framework for Unifying Reordering Transformations (1993) TR

• On the Complexity of Loop Fusion Alain Darte, PACT 1999


M. O’Boyle

Program Transformations

February, 2014
What is a program transformation

- A program transformation is a rewriting of the program such that it has the same semantics

- More conservatively, all data dependences must be preserved

- Previous lectures looked at IR→IR transformations or assembler→assembler transformations

- Focus on transformations in the high level source prog. language: source to source transformations

- Why: Only place where memory reference explicit. Key to restructuring for memory behaviour and large scale parallelism.
Classification

Ongoing open question on a correct taxonomy

• Loop
  – Structure reordering. Change number of loops
  – Iteration reordering. Reorder loop traversal
  – Linear models. Express transformation as unimodular matrices.

• Array
  – Index reordering
  – Duality with loops. Global vs Local.

• All transformations have an associated legality test though some are always legal.
**Loop Restructuring Index Splitting**

Always a legal transformation. No test needed

Do $i = 1, 50$
   $a(101 -i) = a(i)$

Do $i = 1, 100$
   $a(101 -i) = a(i)$
Enddo

Do $i = 51, 100$
   $a(101 -i) = a(i)$
Enddo

A sequential loop with dependence [*] is transformed into two independent parallel loops. Careful selection of split point.

Neither access in each loop refers to same memory location.

All of first loop must execute before second though - why?
Loop Restructuring: Loop Unrolling

Used for exploiting Instruction Level Parallelism

Always legal - take care of epilogue using index splitting

Do $i = 1, 100$

\[
a(i) = i \\
a(i+1) = i+1 \\
a(i+2) = i+2
\]

Enddo

Do $i = 1, 100$

\[
a(i) = i
\]

Enddo

Non-convex iteration space after transformation - steps. Causes difficulties for dependence analysis. Can normalise loop though
### Loop Restructuring: Loop Distribution

<table>
<thead>
<tr>
<th>Loop 1</th>
<th>Loop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Do i = 1,10</code>&lt;br&gt;<code>a(i) = a(i-1)</code>&lt;br&gt;<code>Enddo</code></td>
<td><code>1 2 3 4 10</code>&lt;br&gt;<code>s1</code>&lt;br&gt;<code>s2</code></td>
</tr>
<tr>
<td><code>Do i = 1,10</code>&lt;br&gt;<code>a(i) = a(i-1)</code>&lt;br&gt;<code>Enddo</code></td>
<td><code>1 2 3 4 10</code>&lt;br&gt;<code>s1</code>&lt;br&gt;<code>s2</code></td>
</tr>
</tbody>
</table>

Diagram:

- Loop 1:
  - Index: 1 to 10
  - Variable: `a(i)`
  - Initial value: `a(i-1)`
- Loop 2:
  - Index: 1 to 10
  - Variable: `a(i)`
  - Initial value: `a(i-1)`

Arrows indicate the flow of data between loops.
## Loop Distribution + Statement Reordering

<table>
<thead>
<tr>
<th>Do i = 1,10</th>
<th>a(i) =</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>=a(i+1)</td>
</tr>
<tr>
<td>Enddo</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s2</td>
</tr>
</tbody>
</table>

Do i = 1,10  
=a(i+1)  
Enddo  
Do i = 1,10  
=a(i+1)  
Enddo  
Do i = 1,10  
a(i) =  
Enddo

Anti-dependences honoured.
Loop Restructuring: Loop Fusion

Inverse of loop distribution - needs conformant loops

<table>
<thead>
<tr>
<th>Do i = 1,100</th>
<th>Do i = 1,100</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(i) =</td>
<td>a(i) =</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>Do j = 1,100</td>
<td>Do j = 1,100</td>
</tr>
<tr>
<td>b(j) =</td>
<td>b(i) =</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
</tbody>
</table>

More difficult than distribution. Dependence constrains application.

Used for increasing ILP and improving register use. Also for fork/join based parallelisation.

Loops can be partly fused after pre-distribution
## Iteration reordering: Loop interchange

### Important widely used transformation

<table>
<thead>
<tr>
<th>Do $i = 1, N$</th>
<th>Do $i = 1, N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do $j = 1, N$</td>
<td>Do $j = 1, N$</td>
</tr>
<tr>
<td>$a(i, j) = a(i, j-1) + b(i)$</td>
<td>$a(i, j) = a(i-1, j+1) + b(i)$</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
</tbody>
</table>

$[i, j] \leftrightarrow [j, i]$

Illegal to interchange $[1, -1], [<, >]$ why?
Iteration reordering: Loop interchange

Illegal to interchange [1,-1]: New vector [-1,1]:
Impossible dependence.
Linear models check $TD > 0$
Loop skewing

Always legal used in wavefront parallelisation

\[
\begin{align*}
\text{Do } i &= 1, N \\
\text{Do } j &= 1, N \\
a(i, j) &= a(i, j-1) + b(i) \\
\text{Enddo} \\
\text{Enddo}
\end{align*}
\]

\[
\begin{align*}
\text{Do } i &= 1, N \\
\text{Do } j &= i+1, i+N \\
a(i, j-i) &= a(i, j-i-1) + b(i) \\
\text{Enddo} \\
\text{Enddo}
\end{align*}
\]

- \([i, j] \mapsto [i, j + i]\)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Loop reversal

| Do i = 1, N |
| Do j = 1, N |
| a(i, j) = a(i, j-1) + b(i) Enddo |
| Enddo |

| Do i = N, 1, -1 |
| Do j = 1, N |
| a(i, j) = a(i, j-1) + b(i) Enddo |
| Enddo |

• \([i, j] \mapsto [-i, j]\)

• Rarely used in isolation. In unison with previous two.

• Can combine interchange, shewing and reversal as unimodular transformations/ More on this later.
Tiling = strip-mining plus interchange

\[ \text{Do } i = 1, N, s \]
\[ \text{Do } j = 1, N, s \]
\[ \text{Do } ii = i, i+s-1 \]
\[ \text{Do } jj = j, j+s-1 \]
\[ a(ii,jj) = a(ii,jj) + b(ii) \]
\[ \text{Enddo} \]
\[ \text{Enddo} \]
\[ \text{Enddo} \]

Strip-mine by factor s Non-convex space
Interchange placing smaller strip-mine inside

M. O'Boyle
Program Transformations
February, 2014
Array layout transformations

• Less extensive literature though perhaps have a more significant impact

• Loop transformations affect all memory references within the loop but not elsewhere. Local in nature

• Array and more generally data transformations have global impact but do not affect other references to other arrays.

• Array layout transformations are used to improve memory access performance

• Also form the basis for data distribution based parallelisation schemes for distributed memory machines.
Global index reordering

Dual of loop interchange. Always legal! \([i_i, i_2] \mapsto [i_2, i_1]\)

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i =1, 9</td>
<td>Do i =1, 9</td>
</tr>
<tr>
<td>Do j = 2,20</td>
<td>Do j = 2,20</td>
</tr>
<tr>
<td>a(i,j) = a(i+1,j-1) +b(i)</td>
<td>a(j,i) = a(j-1,i+1) +b(i)</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>a(1,2) =0</td>
<td>a(2,1) =0</td>
</tr>
</tbody>
</table>

- Array declaration and subscripts interchanged globally
- Difficulties occur if array reshaped on procedure boundaries
## Linearisation/delinearisation

Dual of loop strip-mining/linearisation

<table>
<thead>
<tr>
<th>REAL a[10,20]</th>
<th>REAL a[200]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i = 1, 9</td>
<td>Do i = 1, 9</td>
</tr>
<tr>
<td>Do j = 2, 20</td>
<td>Do j = 2, 20</td>
</tr>
<tr>
<td>a(i,j) = a(i+1,j-1) + b(i)</td>
<td>a(20*(i-1)+j) = a(20*(i)+j-1) + b(i)</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>a(1,2) = 0</td>
<td>a(2) = 0</td>
</tr>
</tbody>
</table>
Padding

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i =1, 9</td>
<td>Do i =1, 9</td>
</tr>
<tr>
<td>Do j = 2,20</td>
<td>Do j = 2,20</td>
</tr>
<tr>
<td>a(i,j) = a(i+1,j-1) +b(i)</td>
<td>a(i,j) = a(i+1,j-1) +b(i)</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>Enddo</td>
<td>Enddo</td>
</tr>
<tr>
<td>a(1,2) =0</td>
<td>a(1,2) =0</td>
</tr>
</tbody>
</table>

- Frequently used to overcome cache conflicts. Very simple

- Pad factor 7 in first index. Normally prime.
Unification

- Presentation - simplistic conditions of application can be complex for arbitrary programs.

- Little overall structure.

- Unimodular transformation theory based on linear representation

- Extended to non-singular and the Unified Transformation Framework of Bill Pugh.

- Will return to look in more detail at this formulation in later lectures.
Summary

• Large suite of transformations

• Loop restructuring and reordering

• Legality constraints restrict application

• Array based transformations. Always legal but global impact

• Unifying theories provide structured taxonomy.

• Next lecture: Vectorisation