Compiler Optimisation
9 – Program Transformations

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019
This lecture:

- 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

NB: Simplified presentation.

*Large* number of technicalities.

**Read the book!**
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.

Now, focus on transformations at higher level: source to source transformations.

Why: Only place where memory reference explicit. Key to restructuring for memory behaviour and large scale parallelism.
Introduction
Transformation 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 uni-modal matrices.

- **Array**
  - Index reordering
  - Duality with loops. Global vs Local.

- All transformations have an associated legality test though some a few are always legal.
Loop restructuring
Transformation: index splitting

- A sequential loop with dependence [*] is transformed into two independent parallel loops. Careful selection of split point.
- Always a legal transformation. No test needed

Original
for(i = 1 to 100)
a[101 - i] = a[i]

Split at \(i = 51\)
for(i = 1 to 50)
a[101 - i] = a[i]
for(i = 51 to 100)
a[101 - i] = a[i]

Lots of dependences

- Neither access in each loop refers to same memory location.
- All of first loop must execute before second though - why?
Loop restructuring
Transformation: loop unrolling

- Replicate loop body
- Used for exploiting ILP
- Always a legal transformation. No test needed

<table>
<thead>
<tr>
<th>Original</th>
<th>Unroll 3 times</th>
</tr>
</thead>
</table>
| `for(i = 1 to 100)`  
  `a[i] = i`               | `for(i = 1 to 100 step 3)`  
  `a[i] = i`  
  `a[i+1] = i+1`  
  `a[i+2] = i+2`  
  `for(i = 100 to 100)`  
  `a[i] = i` |

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

**Transformation: loop distribution**

- Move loop statements into their own loops

### Original

```plaintext
for(i = 1 to 10)
    a[i] = S_1
    = a[i-1] S_2
```

### Distributed

```plaintext
for(i = 1 to 10)
    a[i] = S_1

for(i = 1 to 10)
    = a[i-1] S_2
```
Loop restructuring
Transformation: loop distribution + statement reordering

- Anti-dependences honoured

### Original

```plaintext
for(i = 1 to 10)
    a[i] = S_1
    = a[i+1] S_2
```

### Distributed

```plaintext
for(i = 1 to 10)
    = a[i+1] S_2
for(i = 1 to 10)
    a[i] = S_1
```
Loop restructuring
Transformation: loop fusion

- Inverse of loop distribution - needs compatible loops

<table>
<thead>
<tr>
<th>Original</th>
<th>Fused</th>
</tr>
</thead>
</table>
| for(i = 1 to 100)  
a[i] =  
for(j = 1 to 100)  
b[j] = | for(i = 1 to 100)  
a[i] =  
b[i] = |

- 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
Transformation: loop interchange

- Switching the order of nested loops
- Important widely used transformation

Original
for(i = 1 to N)
  for(j = 1 to N)
    a[i,j]=a[i,j-1]+b[i]

Interchanged
for(j = 1 to N)
  for(i = 1 to N)
    a[i,j]=a[i,j-1]+b[i]

[i,j] ↦ [j, i]
Iteration reordering

Transformation: loop interchange

- Switching the order of nested loops
- Important widely used transformation

Original

for(i = 1 to N)
  for(j = 1 to N)
    a[i,j]=a[i-1,j+1]+b[i]

Interchanged

for(j = 1 to N)
  for(i = 1 to N)
    a[i,j]=a[i-1,j+1]+b[i]

[i, j] ↦ [j, i]

Illegal to interchange [1,-1], [<,>] why?
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

**Original**

```latex
\begin{align*}
\text{for}(i = 1 \text{ to } N) \\
&\quad \text{for}(j = 1 \text{ to } N) \\
&\quad a[i,j] = a[i-1,j] + \\
&\quad a[i,j-1]
\end{align*}
```

**Skewed**

```latex
\begin{align*}
\text{for}(i = 1 \text{ to } N) \\
&\quad \text{for}(j = i+1 \text{ to } i+N) \\
&\quad a[i,j-i] = a[i-1,j-i] + \\
&\quad a[i,j-i-1]
\end{align*}
```

- \( [i, j] \mapsto [i, j + i] \)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing
- Used in wavefront parallelisation

Original
```
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+ a[i,j-1]
```

Skewed
```
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+ a[i,j-i-1]
```

- \([i,j] \mapsto [i,j + i]\)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

Original

\[
\begin{align*}
\text{for}(i = 1 \text{ to } N) \\
\ &\text{for}(j = 1 \text{ to } N) \\
\ &\quad a[i,j] = a[i-1,j]+a[i,j-1]
\end{align*}
\]

Skewed

\[
\begin{align*}
\text{for}(i = 1 \text{ to } N) \\
\ &\text{for}(j = i+1 \text{ to } i+N) \\
\ &\quad a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
\end{align*}
\]

\[
[i,j] \mapsto [i,j+i]
\]

- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping.
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

**Original**

```plaintext
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+a[i,j-1]
```

**Skewed**

```plaintext
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
```

- \([i,j] \mapsto [i,j+i]\)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

Original

```
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+ a[i,j-1]
```

Skewed

```
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+ a[i,j-i-1]
```

- $[i,j] \mapsto [i,j+i]$
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

**Original**

```plaintext
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+a[i,j-1]
```

**Skewed**

```plaintext
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
```

- \([i,j] \mapsto [i,j+i]\]
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

Original

```plaintext
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+a[i,j-1]
```

Skewed

```plaintext
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
```

- \([i,j] \mapsto [i,j + i]\)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

Original

```
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+a[i,j-1]
```

Skewed

```
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
```

- $[i,j] \mapsto [i,j+i]$
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing
- Used in wavefront parallelisation

Original
for(i = 1 to N)
  for(j = 1 to N)
    a[i,j] = a[i-1,j]+a[i,j-1]

Skewed
for(i = 1 to N)
  for(j = i+1 to i+N)
    a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]

- \([i,j] \mapsto [i,j + i]\)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing

- Used in wavefront parallelisation

**Original**

```plaintext
for(i = 1 to N)
  for(j = 1 to N)
    a[i,j] = a[i-1,j]+a[i,j-1]
```

**Skewed**

```plaintext
for(i = 1 to N)
  for(j = i+1 to i+N)
    a[i,j-i] = a[i-1,j-i]+a[i,j-i-1]
```

- \( [i,j] \mapsto [i,j+i] \)
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
Iteration reordering
Transformation: loop skewing
- Used in wavefront parallelisation

Original
```
for(i = 1 to N)
    for(j = 1 to N)
        a[i,j] = a[i-1,j]+ a[i,j-1]
```

Skewed
```
for(i = 1 to N)
    for(j = i+1 to i+N)
        a[i,j-i] = a[i-1,j-i]+ a[i,j-i-1]
```

- $[i,j] \mapsto [i,j+i]$
- Equivalent to a change of basis.
- Shifting by a constant referred to as loop bumping
### Iteration reordering

**Transformation: loop reversal**

- Reverse loop direction

<table>
<thead>
<tr>
<th>Original</th>
<th>Fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i = 1 to N)</td>
<td>for(i = N to 1 step -1)</td>
</tr>
<tr>
<td>for(j = 1 to M)</td>
<td>for(j = 1 to M)</td>
</tr>
<tr>
<td>a[i, j] = a[i, j-1]+b[i]</td>
<td>a[i, j] = a[i, j-1]+b[i]</td>
</tr>
</tbody>
</table>

- \([i, j] \mapsto [-i, j]\)

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

- Can combine interchange, skewing and reversal as uni-modal transformations.
Iteration reordering
Transformation: loop tiling/blocking

- Break loop into rectangular tiles
- May increase locality (reduce cache misses)

### Original
for(i = 1 to N)
  for(j = 1 to M)
    a[i,j] = a[i,j]+b[i]

### Tiled
for(i = 1 to N step si)
  for(j = 1 to M step sj)
    for(ii = i to i+si-1)
      for(jj = j to j+sj-1)
        a[ii,jj] = a[ii,jj]+b[ii]

- Non-convex space
- Interchange placing smaller strip-mine inside
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.
Array layout transformations
Transformation: global index reordering

- Swap indices (transpose)
- Dual of loop interchange
- \([i, j] \mapsto [j, i]\)

Original
```c
int a[10,20]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[i,j] = a[i+1,j-1]+b[i]
    a[1,2] = 0
```

Indices reordered
```c
int a[20,10]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[j,i] = a[j-1,i+1]+b[i]
    a[2,1] = 0
```

- Array declaration and subscripts interchanged globally
- Difficulties occur if array reshaped on procedure boundaries
Array layout transformations
Transformation: linearisation

- Map multidimensional array to fewer dimensions (mostly one)
- Dual of loop linearisation

**Original**
```c
int a[10,20]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[i,j] = a[i+1,j-1]+b[i]

a[1,2] = 0
```

**Linearised**
```c
int a[200]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[20*(i-1)+j]=a[20*i+j-i]+b[i]

a[2] = 0
```
Array layout transformations
Transformation: padding

- Increase one or more dimensions with redundant values

Original
```c
int a[10,20]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[i,j] = a[i+1,j-1]+b[i]
a[1,2] = 0
```

Padded by 7
```c
int a[17,20]
for(i = 1 to 9)
    for(j = 2 to 20)
        a[i,j] = a[i+1,j-1]+b[i]
a[1,2] = 0
```

- 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.
- Uni-modular 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

- 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
The biggest revolution in the technological landscape for fifty years

Now accepting applications!

Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk

• 4-year programme: MSc by Research + PhD

• Research-focused: Work on your thesis topic from the start

• Collaboration between:
  ▶ University of Edinburgh’s School of Informatics
    ★ Ranked top in the UK by 2014 REF
  ▶ Edinburgh Parallel Computing Centre
    ★ UK’s largest supercomputing centre

• Research topics in software, hardware, theory and application of:
  ◀ Parallelism
  ◀ Concurrency
  ◀ Distribution

• Full funding available

• Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk