CS415 Compilers

Compiler Optimalizations:
- Vectorization/Parallelization,
- Common Subexpression Elimination,
- Procedure Abstractions

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University.
Review - Data Dependences (stmt./instr. level)

Data Dependences $\Rightarrow$ defined on memory locations / registers and not values

Statement/instruction $b$ data depends on statement/instruction $a$ if there exists:

- **true or flow dependence**
  - $a$ writes a location/register that $b$ later reads (RAW conflict)

- **anti dependence**
  - $a$ reads a location/register that $b$ later writes (WAR conflict)

- **output dependence**
  - $a$ writes a location/register that $b$ later writes (WAW conflict)

<table>
<thead>
<tr>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = a$</td>
<td>$= a$</td>
<td>$a = a$</td>
</tr>
</tbody>
</table>

Lecture 22
A statement $S_1$ **control depends** on statement $S_2$ iff 
(a) $S_2$ computes a conditional branch and  
(b) the execution of $S_1$ depends on this branch (in one case $S_1$ will be 
executed, but not in the other).

\[
\begin{align*}
S_1 & : \text{if } (a == b) \\
S_2 & : a = a + b \\
S_3 & : b = a + b
\end{align*}
\]

Data and control dependences define ORDER CONSTRAINTS that need to be 
respected in order to generate correct code.

**Fundamental Theorem of Dependence**

**Theorem**

Any reordering transformation that preserves 
every dependence in a program preserves the 
meaning of that program.
Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
    a(i) = a(i+1) + 1

Iteration space

anti-dependences with dependence distance = 1
Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
    a(i) = a(i+1) + 1

Iteration space

| 3 | 4 | 5 | ... | 98 | 99 |

anti-dependences with dependence distance = 1

forall i := 3, 99
    a(i) = a(i+1) + 1

a(3:99) = a(4:100) + 1

Loop cannot be parallelized since no order constraints among loop iterations => anti-dep. may be violated!

Loop can be vectorized since RHS is read before LHS is written => anti-dep. is preserved!
Parallelization vs Vectorization

Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

```
array a(1:100) of float
for i := 3, 99
    a(i) = a(i) + 1
```

Iteration space

only intra-statement anti-dependences which are preserved by statement execution semantics

```
forall i := 3, 99
    a(i) = a(i) + 1
```

```
a(3:99) = a(3:99) + 1
```
Parallelization vs Vectorization

Our Goal: Find out whether a loop can be parallelized or vectorized based on data dependence analysis (assume: body of a loop is a basic block)

array a(1:100) of float
for i := 3, 99
    a(i) = a(i-1) + 1

Iteration space

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>98</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

true-dependences with dependence distance = 1

forall i := 3, 99
a(i) = a(i-1) + 1  \[ \times \]

a(3:99) = a(2:98) + 1  \[ \times \]

Loop cannot be parallelized since each array assignment depends on previous array assignment => inherently sequential execution

Loop cannot be vectorized
Perfectly nested loop, \( A \) is \( m \)-dimensional array

\[
\begin{align*}
\text{DO } i_1 &= L_1, U_1 \\
\text{DO } i_2 &= L_2, U_2 \\
&\quad \ldots \\
\text{DO } i_n &= L_n, U_n \\
S_1 &= A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \\
S_2 &= \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)) \\
\text{ENDDO} \\
&\quad \ldots \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[ f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \leq i \leq m \]
\[
\alpha < \beta
\]

Finding integer solutions for this set of simultaneous equations with constraints is an integer programming problem, which is NP-complete.
Our assumptions:
- Singly nested loop
- Single assignment statement
- All arrays are one-dimensional
- No aliasing among arrays
- \( f(i) \) is an affine function of the induction variable \( i \): \( a \times i + c \), with a positive integer \( a > 0 \), and \( c \) an integer

\[ \Rightarrow \text{Anti and output dependences don't matter here for vectorization} \]

Goal of dependence testing:
1. Prove that true dependence does not exist;
2. If cannot do that, show that dependence exists with a fixed distance vector;
3. If cannot do that, assume existence of dependence

Dependence testing algorithm:
For each pair of LHS and RHS array references \( <X(f(i)), Y(g(i))> \) do perform dependence test

for \( i := \text{lb}, \text{ub} \)

\[ X(f(i)) = \ldots Y(g(i)) \ldots \]
Data Dependence Testing Algorithm

for i := lb, ub
    \( X(f(i)) = \ldots Y(g(i)) \ldots \)

**Input:** \(<X(f(i)), Y(g(i))>\)

**Output:** no dependence, dependence with distance vector \( d \), or dependence

**Method:** (cascading tests)

- if \( X \neq Y \) then report “no dependence” else
  - if \( f(i) = i + c \) and \( g(i) \) is a constant (or visa versa) then
    apply simple ZIV test else
      - if \( f(i) = a \times i + c_1 \) and \( g(i) = a \times i + c_2 \) then
        apply strong SIV test else
          report “dependence”
for $i := lb, ub$

$X(f(i)) = \ldots X(g(i)) \ldots$

$f(i) = i + c_1$ and $g(i) = c_2$, i.e., $g(i)$ is a constant; this means that the same memory location $X(c_2)$ is read in all iterations.

Examples:

$\langle A(i), A(5) \rangle$

$\langle A(i+1), A(7) \rangle$

No dependence exists if $(i + c_1 - c_2) \neq 0$ for all $lb \leq i \leq ub$:

$lb \leq |c_1 - c_2| \leq ub \Rightarrow$ dependence

Note: This is a special case of the weak-zero SIV test.
for $i := \text{lb, ub}$

\[ X(f(i)) = \ldots X(g(i)) \ldots \]

$f(i) = a \times i + c_1$ and $g(i) = a \times i + c_2$

Examples:

\[
\begin{align*}
\langle A(i), A(i - 1) \rangle \\
\langle A(4i + 2), A(4i - 1) \rangle
\end{align*}
\]

Dependence exists with distance $d$ if there exists an integer solution to the following equation:

\[
- f(i) = a \times i + c_1 \text{ and } g(i') = a \times i' + c_2
\]

\[
- d = i' - i = \frac{c_1 - c_2}{a}
\]

and $d$ has to be (1) a positive integer and (2) $d \leq (\text{ub-\text{lb}})$

- if dependence exists, report “dependence with distance $d$”
for $i := lb, ub$

$X(f(i)) = ... X(g(i)) ...$

$d = i' - i = \frac{c_1 - c_2}{a}$

$d \leq ub - lb$
for i := 1, 100
    a(i) = a(i) + 1

When can we vectorize this for loop?
    a(1:100) = a(1:100) +1

How do we say this in ILOC?
    - use vecton and vectoff instructions
ILOC - Loop Implementation Code

for (i := 1, 100) { assignment }
next statement

"sequential" loop code

set vector execution
reset vector execution

branch $L_x$

$L_0$: loadI 1 $\Rightarrow r_1$
loadI 1 $\Rightarrow r_2$
loadI 100 $\Rightarrow r_3$
cmp_GE $r_1, r_3 \Rightarrow r_4$
cbr $r_4 \Rightarrow L_2, L_1$

$L_1$: assignment
add $r_1, r_2 \Rightarrow r_1$
cmp_LT $r_1, r_3 \Rightarrow r_5$
cbr $r_5 \Rightarrow L_1, L_2$
branch $L_2$

$L_x$: vecton or no-op
branch $L_0$

$L_2$: vectoff
next statement
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \[ a(i) * a(i) \]

\[
\begin{align*}
4. & \quad t_1 = \text{addr}(a) - 4 \\
5. & \quad t_2 = i * 4 \\
6. & \quad t_3 = t_1[t_2] \\
\ldots
\end{align*}
\]
\[ a(i) \times a(i) \]

4. \[ t_{1} = \text{addr}(a) - 4 \]
5. \[ t_{2} = i \times 4 \]
6. \[ t_{3} = t_{1}[t_{2}] \]
7. \[ t_{4} = \text{addr}(a) - 4 \]
8. \[ t_{5} = i \times 4 \]
9. \[ t_{6} = t_{4}[t_{5}] \]
10. \[ t_{7} = t_{3} \times t_{6} \]

[Diagram of Basic Block DAG Construction]

- \[ t_{1}, t_{4} \]
- \[ i \]
- \[ 4 \]
- \[ t_{2}, t_{5} \]
- \[ t_{3}, t_{6} \]
- \[ *, t_{7} \]
- \[ [\ ], t_{3}, t_{6} \]
Local common subexpression elimination (CSE):

\[ a(i) \times a(i) \]

4. \[ t1 = \text{addr}(a) - 4 \]
5. \[ t2 = i \times 4 \]
6. \[ t3 = t1[t2] \]
7. \[ t4 = \text{addr}(a) - 4 \]
8. \[ t5 = i \times 4 \]
9. \[ t6 = t4[t5] \]
10. \[ t7 = t3 \times t6 \]

\[ \text{code generated:} \]

\[ t1 = \text{addr}[a]-4 \]
\[ t2 = i \times 4 \]
\[ t3 = t1[t2] \]
\[ t7 = t3 \times t3 \]
How to add a subexpression into a partially constructed DAG:

\[ A = B + C \]

Is there a node already for \( B + C \)?
- If so, add \( A \) to its list of labels.
- If not:
  - is there a node labeled \( B \) already?
    - If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already?
    - If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for +, with left child \( B \) and right child \( C \).

How to do this? HASHING \(<op, node(opd1), node(opd2)\>"
How to add a subexpression into a partially constructed DAG:

\[
A = B + C
\]

Is there a node already for \( B + C \)? \(<+, \text{node}(B), \text{node}(C)> \) defined?
- If so, add \( A \) to its list of labels.
- If not:
  - is there a node labeled \( B \) already? \( \text{node}(B) \) defined?
    If not, create a leaf labeled \( B \).
  - Is there a node labeled \( C \) already? \( \text{node}(C) \) defined?
    If not, create a leaf labeled \( C \).
  - Create a node labeled \( A \), for \(+\), with left child \( B \) and right child \( C \).

How to do this? HASHING \(<\text{op}, \text{node}(\text{opd1}), \text{node}(\text{opd2})>\)
DAG Construction Algorithm

Summary:
- every expression is assigned a value number
  examples: node(a),
  node(4),
  node(<+, valNum1, ValNum2>)
- assignment changes value number associated with LHS variable

- implementation of value numbers
  • use pointers of nodes in DAG
  • use virtual register numbers (code shape encoding!)

You could do this in a single pass in our compiler!
Procedure abstraction

Read EaC: Chapter 6.1 - 6.5