Announcements

• Project 2 deadline extension: Friday, November 30.

• Homework 5 will be posted by Wednesday, November 28.

• Project 3 will be posted by Friday, November 30. Due on Friday, December 14 (last day of reading period).

• final exam:
  Tuesday, December 18, noon to 3:00pm
  ⇒ any conflicts?
Shared Memory Programming with OpenMP

- MIMD architecture (multiple instructions, multiple data)
- Allows expression of parallelism at different levels: task and loop level. Parallelization through **pragmas**.
- Basic fork/join thread execution model with barrier synchronization between parallel regions.
Two important issues while specifying the parallel execution of a `for` loops:

- **safety** – parallel execution has to preserve all dependences

- **profitability** – benefits of parallel execution have to compensate for the overhead penalty, both in terms of execution time and energy consumption

Target platforms: NVIDIA Jetson TK-1, TX-1, and TX-2 boards
dependence relation: Describes all statement-to-statement execution orderings for a sequential program that must be preserved if the meaning of the program is to remain the same.

There are two sources of dependences:

**data dependence**

\[
S_1 \quad \text{pi} = 3.14 \\
S_2 \quad r = 5.0 \\
S_3 \quad \text{area} = \text{pi} \times r^{**2}
\]

**control dependence**

\[
S_1 \quad \text{if (t .ne. 0.0) then} \\
S_2 \quad a = a/t \\
\text{endif}
\]

How to preserve the meaning of these programs?
Execute the statements in an order that preserves the original load/store order.
Dependence — Basics

Theorem
Any reordering transformation that preserves every dependence (i.e., visits first the source, and then the sink of the dependence) in a program preserves the meaning of that program.

Note: Dependence starts with the notion of a sequential execution, i.e., starts with a sequential program.
**Dependence — Overview**

**Definition** — There is a data dependence from statement $S_1$ to statement $S_2$ ($S_1 \delta S_2$) if

1. Both statements access the same memory location, and
2. There is a run–time execution path from $S_1$ to $S_2$.

**Data dependence classification**

“$S_2$ depends on $S_1$” — $S_1 \delta S_2$

- **true (flow) dependence**
  
  occurs when $S_1$ writes a memory location that $S_2$ later reads

- **anti dependence**
  
  occurs when $S_1$ reads a memory location that $S_2$ later writes

- **output dependence**
  
  occurs when $S_1$ writes a memory location that $S_2$ later writes

- **input dependence**
  
  occurs when $S_1$ reads a memory location that $S_2$ later reads. Note: Input dependences do not restrict statement (load/store) order!
Dependence — Where do we need it?

We restrict our discussion to data dependence for scalar and subscripted variables (no pointers and no control dependence).

Examples:

$$\begin{align*}
\text{do } I = 1, 100 & \quad \text{do } I = 1, 99 \\
\text{do } J = 1, 100 & \quad \text{do } J = 1, 100 \\
& \quad A(I, J) = A(I, J) + 1 \quad A(I, J) = A(I+1, J) + 1 \\
& \quad \text{endo} \quad \text{endo} \\
& \quad \text{endo} \quad \text{endo}
\end{align*}$$

vectorization

$$\begin{align*}
A(1:100:1,1:100:1) &= A(1:100:1,1:100:1) + 1 \\
A(1:99,1:100) &= A(2:100,1:100) + 1
\end{align*}$$

parallelization

$$\begin{align*}
\text{doall } I = 1, 100 & \quad \text{do } I = 1, 99 \\
\text{doall } J = 1, 100 & \quad \text{doall } J = 1, 100 \\
& \quad A(I, J) = A(I, J) + 1 \quad A(I, J) = A(I+1, J) + 1 \\
& \quad \text{endo} \quad \text{endo} \\
& \quad \text{ implicit barrier sync. } \quad \text{implicit barrier sync.} \\
& \quad \text{endo} \quad \text{endo} \\
& \quad \text{ implicit barrier sync. }
\end{align*}$$
**Dependence Analysis**

**Question**

Do two variable references never/maybe/always access the same memory location?

**Benefits**

- improves alias analysis
- enables loop transformations

**Motivation**

- classic optimizations
- instruction scheduling
- data locality (register/cache reuse)
- vectorization, parallelization

**Obstacles**

- array references
- pointer references
Vectorization vs. Parallelization

**vectorization** — Find parallelism in innermost loops; fine–grain parallelism

**parallelization** — Find parallelism in outermost loops; coarse–grain parallelism

- Parallelization is considered more complex than vectorization, since finding coarse–grain parallelism requires more analysis (e.g., interprocedural analysis).

- Automatic vectorizers have been very successful
A **loop-independent** dependence exists regardless of the loop structure. The source and sink of the dependence occur on the same loop iteration.

A **loop-carried** dependence is induced by the iterations of a loop. The source and sink of the dependence occur on different loop iterations.

*Loop-carried dependences can inhibit parallelization and loop transformations*
Dependence Testing

Given

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

A dependence between statement \( S_1 \) and \( S_2 \), denoted \( S_1\delta S_2 \), indicates that \( S_1 \), the source, must be executed before \( S_2 \), the sink on some iteration of the nest.

Let \( \alpha \) \& \( \beta \) be a vector of \( n \) integers within the ranges of the lower and upper bounds of the \( n \) loops.

\[
\text{Does } \exists \alpha \leq \beta, \text{ s.t. } f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m ?
\]
Iteration Space

\[
\begin{array}{c}
do \ I = 1, 5 \\
\hspace{1cm} do \ J = I, 6 \\
\hspace{2cm} \ldots \\
\hspace{2cm} \text{enddo} \\
\text{enddo} \\
1 \leq I \leq 5 \\
I \leq J \leq 6 \\
\end{array}
\]

- lexicographical (sequential) order for the above iteration space is

\[
(1,1), (1,2), \ldots, (1,6) \\
(2,2), (2,3), \ldots (2,6) \\
\ldots \\
(5,5), (5,6)
\]

- given \( I = (i_1, \ldots i_n) \) and \( I' = (i'_1, \ldots, i'_n) \),

\[
I < I' \text{ iff } \\
(i_1, i_2, \ldots i_k) = (i'_1, i'_2, \ldots i'_k) \quad \& \quad i_{k+1} < i'_{k+1}
\]
Distance & Direction Vectors

\begin{align*}
d & \text{do I = 1, N} \\
& \text{do J = 1, N} \\
S_1 & A(I,J) = A(I,J-1) \\
& \text{enddo} \\
& \text{enddo} \\
S_2 & A(I,J) = A(I-1,J-1) \\
S_3 & B(I,J) = B(I-1,J+1) \\
& \text{enddo} \\
& \text{enddo}
\end{align*}

Distance Vector = number of iterations between accesses to the same location

Direction Vector = direction in iteration space (=, <, >)

distance vector direction vector

\begin{align*}
S_1 \delta S_1 \\
S_2 \delta S_2 \\
S_3 \delta S_3
\end{align*}
Which Loops are Parallel?

\[
\begin{align*}
S_1 & \quad A(I,J) = A(I,J-1) \\
S_2 & \quad A(I,J) = A(I-1,J-1) \\
S_3 & \quad B(I,J) = B(I-1,J+1)
\end{align*}
\]

- 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 \( \nexists \) a dependence \( D_j \) carried at level \( i \)

<table>
<thead>
<tr>
<th></th>
<th>distance vector</th>
<th>direction vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall D_j )</td>
<td>( d_1, \ldots, d_{i-1} &gt; 0 )</td>
<td>( d_1, \ldots, d_{i-1} = &quot;&lt;&quot; )</td>
</tr>
<tr>
<td>OR</td>
<td>( d_1, \ldots, d_i = 0 )</td>
<td>( d_1, \ldots, d_i = &quot;=&quot; )</td>
</tr>
</tbody>
</table>
Approaches to Dependence Testing

- can we solve this problem exactly?
- what is conservative in this framework?
- restrict the problem to consider index and bound expressions that are linear functions

$\Rightarrow$ solving general system of linear equations in integers is NP-hard

Solution Methods

- inexact methods
  - Greatest Common Divisor (GCD)
  - Banerjee’s inequalities

- cascade of exact, efficient tests
  (fall back on inexact methods if needed)
  - Rice (see posted PLDI’91 paper)
  - Stanford

- exact general tests (integer programming)
Dependence Testing

SIV - Single Induction Variable Test

1. Single loop nest with constant lower (LB) and upper (UB) bounds, and step 1

   \[
   \text{for } i = \text{LB, UB, 1} \\
   \text{...} \\
   \text{endfor}
   \]

   The loop bounds define the iteration space for loop induction variable \( i \).

2. Two array references with array subscript (index) expressions of the form (true dependence)

   \[
   \text{for } i = \text{LB, UB, 1} \\
   \text{R1: } X(a*i + c1) = ... \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \}
Dependence Testing

There is a dependence between R1 and R2 iff

$$\exists i, i' : i \leq i' \text{ and } (a \ast i + c_1) = (a \ast i' + c_2)$$

where $i$ and $i'$ are two iterations in the iteration space of the loop. This means that in both iterations, the same element of array $X$ would be accessed.

So let’s just solve the equation:

$$\begin{align*}
(a \ast i + c_1) &= (a \ast i' + c_2) \\
\implies \frac{c_1 - c_2}{a} &= i' - i = \Delta d
\end{align*}$$

There is a dependence with distance $\Delta d$ iff

1. $\Delta d$ is an integer value and
2. $UB - LB \geq \Delta d \geq 0$
Dependence Testing Examples

1. 
   for i = LB, UB, 1
   R1: X(i) = ... \ \ write
   R2: ... X(i - 2) ... \ \ read
   endfor

   \[ a = 1, \ c_1 = 0, \ c_2 = -2 \Rightarrow \Delta d = 2 \ (dependence) \]

2. 
   for i = LB, UB, 1
   R1: X(2*i) = ... \ \ write
   R2: ... X(2*i - 1) ... \ \ read
   endfor

   \[ a = 2, \ c_1 = 0, \ c_2 = -1 \Rightarrow \Delta d = \frac{1}{2} \ (no \ dependence) \]

Assume R1 executes before R2.

Classification of dependences:

- R1 is write, R2 is read \( \Rightarrow \) true dependence
- R1 is read, R2 is write \( \Rightarrow \) anti dependence
- R1 is write, R2 is write \( \Rightarrow \) output dependence
Dependence Testing

ZIV - Zero Induction Variable Test

Two array references with array subscript (index) expressions of the form of a constant:

\[
\text{for } i = LB, UB, 1 \\
R1: \ X(c1) = \ldots \quad \text{\textbackslash \ write} \\
R2: \quad \ldots \ X(c2) \ldots \quad \text{\textbackslash \ read} \\
\text{endfor}
\]

where \(c1\), and \(c2\) are integer constants, and R1 and R2 are references to the same array.

There is a dependence between R1 and R2 if\(c1 = c2 = c\).

What is the dependence distance \(\Delta d\)?

Since every iteration \(i\) writes \(X(c)\), and every iteration \(i'\) reads \(X(c)\), there is no fixed distance \(\Delta d\). In fact, both references have true, anti, and output dependences:

\[
\Delta d \in \{0, \ldots UB - LB\} \text{ for true} \\
\Delta d \in \{1, \ldots UB - LB\} \text{ for anti and output}
\]
Loop Transformations

Goal

• modify execution order of loop iterations
• preserve data dependence constraints

Motivation

• data locality
  increase reuse of registers, cache
• parallelization / vectorization
  eliminate or move loop-carried deps, granularity:
  outer (parallel) vs. inner (vector)

Taxonomy

• loop interchange
  (change order of loops in nest)
• loop fusion
  (merge bodies of adjacent loops)
• loop distribution
  (split body of loop into adjacent loops)
• strip-mine and interchange (tiling, blocking)
  (split loop into nested loops, then interchange)
Loop Interchange

\[
\begin{align*}
&\text{do } I = 1, N \\
&\quad \text{do } J = 1, N \\
&S_1 \quad A(I, J) = A(I, J-1) \\
&S_2 \quad B(I, J) = B(I-1, J-1) \\
&\quad \text{enddo} \\
&\quad \text{enddo}
\end{align*}
\]

⇒ loop interchange ⇒

\[
\begin{align*}
&\text{do } J = 1, N \\
&\quad \text{do } I = 1, N \\
&S_1 \quad A(I, J) = A(I, J-1) \\
&S_2 \quad B(I, J) = B(I-1, J-1) \\
&\quad \text{enddo} \\
&\quad \text{enddo}
\end{align*}
\]

Loop interchange is safe \textit{iff}

- it does not create a lexicographically negative direction vector \((1,-1) \rightarrow (-1,1)\)

⇒ Benefits

- may expose parallel/vector loops, incr granularity
- reordering iterations may improve reuse
Loop Fusion

\[
\text{do } i = 2, N \\
S_1 \quad A(i) = B(i) \\
\text{do } i = 2, N \\
S_2 \quad B(i) = A(i-1)
\]

⇒ loop fusion ⇒

\[
\text{do } i = 2, N \\
S_1 \quad A(i) = B(i) \\
S_2 \quad B(i) = A(i-1)
\]

Loop fusion is safe iff

• no loop-independent dependence between nests is converted to a backward loop-carried dep

(would fusion be safe if \(S_2\) referenced \(a(i+1)\) ?)

⇒ Benefits

• reduces loop overhead
• improves reuse between loop nests
• increases granularity of parallel loop
• not useful for vectorization
Loop Distribution (Fission)

\[
\begin{align*}
&\text{do } i = 2, N \\
&S_1 \quad A(i) = B(i) \\
&S_2 \quad B(i) = A(i-1)
\end{align*}
\]

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

\[
\begin{align*}
&\text{do } i = 2, N \\
&S_1 \quad A(i) = B(i) \\
&\text{do } i = 2, N \\
&S_2 \quad B(i) = A(i-1)
\end{align*}
\]

Loop distribution is safe iff

- statements involved in a cycle of true deps (recurrence) remain in the same loop, and
- if \(\exists\) a dependence between two statements placed in different loops, it must be forward

\[\Rightarrow\text{Benefits}\]

- necessary for vectorization
- may enable partial/full parallelization
- may enable other loop transformations
- may reduce register/cache pressure
Data Locality

Why locality?
- memory accesses are expensive
- exploit higher levels of memory hierarchy by reusing registers, cache lines, TLB, etc.
- locality of reference ⇔ reuse

Locality
- temporal locality  
  reuse of a specific location
- spatial locality  
  reuse of adjacent locations
  (cache lines, pages)

What locality/reuse occurs in this loop nest?

\[
\begin{align*}
\text{do } & i = 1, N \\
\text{do } & j = 1, M \\
& A(i) = A(i) + B(j)
\end{align*}
\]
Strip-Mine and Interchange (Tiling)

\[
\begin{align*}
\text{do } i &= 1, N \\
\text{do } j &= 1, M \\
A(i) &= A(i) + B(j) \\
\Rightarrow \text{ Strip Mine} \Rightarrow \\
\text{do } i &= 1, N \\
\text{do } jj &= 1, M, T \\
\text{do } j &= jj, jj+T-1 \\
A(i) &= A(i) + B(j) \\
\Rightarrow \text{ Interchange} \Rightarrow \\
\text{do } jj &= 1, M, T \\
\text{do } i &= 1, N \\
\text{do } j &= jj, jj+T-1 \\
A(i) &= A(i) + B(j)
\end{align*}
\]

Strip mining is always safe, with interchange it

- changes shape of iteration space
- can exploit reuse for multiple loops
Loop Transformations to Improve Reuse

Assumptions

cache architecture (simple):
– one word cache lines,
– LRU replacement policy,
– fully associative cache,
– $M >$ cache size

Analysis

Original loop nest

A(i) — $N$ cache misses, one for each outer iteration (cold misses); reuse for inner iterations

B(j) — $N \cdot M$ cache misses due to LRU policy (capacity misses)
Loop Transformations to Improve Reuse

Transformed loop nest — strip mining and interchange

A(i) — N*M/T cache misses (conservative)

B(j) — M cache misses; once element is in cache, it stays in cache until all computation using it is done

Comparison

\[ N + N \times M \text{ misses} \quad \text{vs.} \quad M + \frac{N \times M}{T} \text{ misses} \]

⇒ tradeoff decision

Note:

• strip–mine and interchange is similar to unroll–and–jam

• typically, cache architectures are more complex

• scalar replacement transformation for registers
A Simple Vectorizing Compiler

How to vectorize the following loops?

for (i=2; i<100; i++) {
    S1: \( a[i] = b[i+1] + 1; \)
    S2: \( b[i] = a[i] + 5; \)
}

for (i=2; i<100; i++) {
    S1: \( a[i] = b[i-1] + a[i-1] + 3; \)
    S2: \( b[i] = a[i+1] + 5; \)
}

Simple vectorizer assumptions:

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array variables
4. simple array index expressions of induction variable \((i +/- c \text{ or } c);\) can use ZIV or SIV test
5. no function calls
A Simple Vectorizing Source-to-Source Compiler

SKETCH OF BASIC ALGORITHM

Here is a basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more

2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do

   (a) if SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential

   (b) if SCC is a single statement and has no dependence cycle (ignore anti), distribute loop around it and generate vector code; otherwise, mark distributed loop sequential.
A more complex example

EXAMPLE

for (i=2; i<99; i++) {
    S1: a[i] = b[i-1] + c[i-1] + 3;
    S2: b[i] = (c[i] + b[i+1]) / 2;
    S3: c[i] = a[i] + 1;
    S4: d[i] = b[i] + c[i+1];
}

STATEMENT-LEVEL DEPENDENCE GRAPH?
A more complex example

for (i=2; i<99; i++) {
    S1: a[i] = b[i-1] + c[i-1] + 3;
    S2: b[i] = (c[i] + b[i+1]) / 2;
    S3: c[i] = a[i] + 1;
    S4: d[i] = b[i] + c[i+1];
}

Partition of dependence graph nodes into SCCs:
{ {S1, S3}, {S2}, {S4} }
A more complex example

Dependencies between SCCs of statement-level dependence graph. This has to be an acyclic graph:

Generated code:

S2: \( b[2:99] = (c[2:99] + b[3:100]) / 2; \)
    for (i=2; i<99; i++) {
        S1: \( a[i] = b[i-1] + c[i-1] + 3; \)
        S3: \( c[i] = a[i] + 1; \)
    }
Other Useful Loop / Data Transformations

1. privatization (parallel) or scalar expansion (vector)

   for i = LB, UB, 1  for i = LB, UB, 1  forall i = LB, UB, 1
   S1:   temp = ...  temp[i] = ...  temp_private = ...
   S2:   ... temp . ... temp[i] ... ... temp_private
     endfor

2. loop peeling

3. loop skewing
   Used to allow wave-front style computations

4. loop splitting
   For crossing dependence thresholds
Project and OpenMP

safety

Sample code:

```c
#pragma omp parallel for private(i, hash)
    for (j = 0; j < num_hf; j++) {
        for (i = 0; i < wl_size; i++) {
            hash = hf[j] (get_word(wl, i));
            hash %= bv_size;
            bv[hash] = 1;
        }
    }
```

This specifies:

- outermost (j-loop) is parallel
- each thread will get its own copy of variables `i` and `hash`, eliminating loop carried anti and output dependences.
Project and OpenMP

**profitability (execution time / energy)**

Sample code:

```c
#define CHUNK_SIZE 2
int chunk = CHUNK_SIZE
#pragma omp parallel for \
  schedule (dynamic, chunk) \
  private(i, hash)
  for (j = 0; j < num_hf; j++) {
    for (i = 0; i < wl_size; i++) {
      hash = hf[j] (get_word(wl, i));
      hash %= bv_size;
      bv[hash] = 1; }
  }
```

This specifies:

- outermost (j-loop) is parallel, with CHUNK_SIZE iterations scheduled as a group; default chunk size=1
- three basic scheduling strategies: static, dynamic, or guided
Next Lecture

Things to do:

• Work on project

• Logic Programming and Prolog