Announcements

• Homework 5 has been posted.

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

• final exam:
  Tuesday, December 18, noon to 3:00pm does not work. What are alternatives?
  Monday, December 17, noon to 3:00pm?
  Wednesday, December 19, noon to 3:00pm?
  ⇒ any conflicts?
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

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

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.

Does $\exists \ \alpha \leq \beta$, s.t.

\[f_k(\alpha) = g_k(\beta) \quad \forall k, \ 1 \leq k \leq m\]
Distance & Direction Vectors

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

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

distance vector direction vector

$S_1 \delta S_1$

$S_2 \delta S_2$

$S_3 \delta S_3$
Which Loops are Parallel?

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

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

<table>
<thead>
<tr>
<th>( \forall D_j )</th>
<th>distance vector</th>
<th>direction vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_1, \ldots, d_{i-1} &gt; 0 )</td>
<td>( d_1, \ldots, d_{i-1} ) = “&lt;”</td>
<td></td>
</tr>
<tr>
<td>( d_1, \ldots, d_i = 0 )</td>
<td>( d_1, \ldots, d_i ) = “=”</td>
<td></td>
</tr>
</tbody>
</table>
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: \( c[i] = a[i] + 1; \)
    }
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

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

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

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

Loop interchange is safe iff

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

\[\Rightarrow \text{ Benefits} \]

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

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

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

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

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)

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

⇒ loop distribution ⇒

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

Loop distribution is safe iff

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

⇒ 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 \(\text{reuse of a specific location}\)
- spatial locality \(\text{reuse of adjacent locations (cache lines, pages)}\)

What locality/reuse occurs in this loop nest?

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

\[
\begin{align*}
&\text{do } i = 1, N \\
&\hspace{1em} \text{do } j = 1, M \\
&\hspace{2em} A(i) = A(i) + B(j) \\
&\implies \text{Strip Mine} \\
&\text{do } i = 1, N \\
&\hspace{1em} \text{do } jj = 1, M, T \\
&\hspace{2em} \text{do } j = jj, jj+T-1 \\
&\hspace{3em} A(i) = A(i) + B(j) \\
&\implies \text{Interchange} \\
&\text{do } jj = 1, M, T \\
&\hspace{1em} \text{do } i = 1, N \\
&\hspace{2em} \text{do } j = jj, jj+T-1 \\
&\hspace{3em} A(i) = A(i) + B(j)
\end{align*}
\]

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,
  - $M > \text{cache size}$

Analysis

Original loop nest

\[ A(i) \rightarrow N \text{ cache misses, one for each outer iteration (cold misses); reuse for inner iterations} \]

\[ B(j) \rightarrow N \times M \text{ cache misses due to LRU policy (capacity misses)} \]
Loop Transformations to Improve Reuse

Transformed loop nest — strip mining and interchange

A(i) — $N \times M / T$ 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

$$N + N \times M \text{ misses vs. } M + \frac{N \times M}{T} \text{ misses}$$

$\Rightarrow$ tradeoff decision

Note:

- strip–mine and interchange is similar to unroll–and–jam
- typically, cache architectures are more complex
- scalar replacement transformation for registers
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 example, for crossing dependence thresholds

5. loop blocking (strip-mining) for partial parallelism

   for i = LB, UB, 1
   a(i) = a(i-10) + 5
   endfor

   for i = LB, UB, 10
   for ii = i, i+9, 1
      a(ii) = a(ii-10) + 5
      endfor
   endfor
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.
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
Compiling for Distributed–Memory Multiprocessors

Assumptions

- Data parallelism has been specified via data layout;
  Example
  
  \textit{align A(i) with B(i)}
  \textit{distribute A(block) on 4 procs}

- Compiler generates SPMD code (single program, multiple data)

- Compiler uses \textit{owner–computes} rule

Compiler challenges

- Has to manage local name spaces (there is no global name space!)

- Insert necessary communication

- \textit{Goal}: Minimize cost of communication while maximizing parallelism
Compiling for Distributed–Memory Multiprocessors

Example

```plaintext
real A(100), B(100)

do i = 1, 99
    A(i) = A(i+1) + B(i+1)
endo
```

![Diagram of array distribution across processors](image)

real A(26), B(26) // 1 element overlap to the right
if my$proc == 3 then my$up=24 else my$up=25 endif
if my$proc > 0 then
    send( A(1), my$proc - 1) //send to left
    send( B(1), my$proc - 1) //send to left
endif
if my$proc < 3 then
    receive( A(26), my$proc + 1) //receive from right
    receive( B(26), my$proc + 1) //receive from right
endif

do i = 1, my$up
    A(i) = A(i+1) + B(i+1)
endo
CUDA – Programming GPUs

See additional slides on our web site.
Next Lecture

Things to do:

- Work on project
- Logic Programming and Prolog