COMP 515: Advanced Compilation for Vector and Parallel Processors

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Announcements

- **Make-up lecture tomorrow (March 5th) in DH 3076 at 10am**

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/

Review

- **Last time, we covered several techniques:**
  - **Scalar Replacement**
  - **Register-Pressure Moderation**
  - **Unroll-and-Jam**
- **All of these techniques involved the register allocater in array references.**

Loop Interchange

- The order of a loop nest effect the effectiveness of register optimization.

Original:

```
DO I = 2, N
   DO J = 1, M
      A(J,I) = A(J,I-1)
   ENDDO
ENDDO
```

```
DO J = 1, M
   DO I = 2, N
      A(J,I) = A(J,I-1)
   ENDDO
ENDDO
```

Optimized:

```
DO I = 2,N
   DO J = 1, M
      R1 = A(J,I-1)
      A(J,I) = R1
   ENDDO
ENDDO
```

```
DO J = 1,M
   R1 = A(J,1)
   DO I = 2,N
      A(J,I) = R1
   ENDDO
ENDDO
```

Considerations

- We want loops that will carry dependence at innermost position.
- Pick loop at level i such that there's a dependence D with:

$$D_j = \begin{cases} < & j = i \\ = & j \neq i \end{cases}$$

Example

```
DO J = 1,N
   DO K = 1,N
     DO I = 1,256
        A(I,J,K) = A(I,J-1,K) +&
       A(I,J-1,K-1) + A(I,J,K-1)
     ENDDO
   ENDDO
ENDDO
```

Dependence Matrix

$$\begin{bmatrix} < & = & = \\ < & < & = \\ = & < & = \end{bmatrix}$$

```
DO K = 1,N
   DO I = 1,256
      DO J = 1,N
         A(I,J,K) = A(I,J-1,K) +&
        A(I,J-1,K-1) + A(I,J,K-1)
      ENDDO
   ENDDO
ENDDO
```

- This eliminated a load under scalar replacement.
- Can continue to exchange I and K loops.

Profitability

- Prefer interchanges that save the most memory operations.
- For each loop L, let profit(L) be the product of L's iteration count and the number of dependency rows that are '=' everywhere but L's position.
- Pick a loop L that maximizes profit(L).

Loop Fusion

- **Consider the following:**

```
DO I = 1,N
   A(I) = C(I) + D(I)
ENDDO

DO I = 1,N
   B(I) = C(I) - D(I)
ENDDO
```

If we fuse these loops, we can reuse operations in registers:

```
DO I = 1,N
   A(I) = C(I) + D(I)
   B(I) = C(I) - D(I)
ENDDO
```

Example

```
DO J = 1,N
  DO I = 1,M
     A(I,J) = C(I,J)+D(I,J)
  ENDDO
  DO I = 1,M
     B(I,J) = A(I,J-1)-E(I,J)
  ENDDO
ENDDO
```

Fusion and Interchange

```
DO I = 1,M
  DO J = 1,N
     B(I,J) = A(I,J-1)-E(I,J)
     A(I,J) = C(I,J)+D(I,J)
  ENDDO
ENDDO
```

Scalar Replacement

```
DO I = 1,M
   r = A(I,0)
   DO J = 1,N
      B(I,J) = r - E(I,J)
      r = C(I,J) + D(I,J)
      A(I,J) = r
   ENDDO
ENDDO
```

We've saved (n-1)m loads

Profitability

- Three categories of loop independent dependencies:
  - Fusion preventing
  - Causes a loop-independent dependence after fusion
  - Causes a forward carried dependence after fusion
- First case is called blocking dependence.
- Second case is called profitable dependence.

Alignment

- **Blocking dependencies can be overcome using loop alignment.**

```
DO I = 1,M
   DO J = 1,N
     A(J,I) = B(J,I)+1.0
   ENDDO
   DO J = 1,N
     c(J,I) = A(J+1,I)+2.0
   ENDDO
ENDDO
```

```
DO I = 1,M
   DO J = 0,N-1
     A(J+1,I) = B(J+1,I)+1.0
   ENDDO
   DO J = 1,N
     c(J,I) = A(J+1,I)+2.0
   ENDDO
ENDDO
```

Alignment

- We can now combine peeling and fusion to get:

```
DO I = 1,M
      A(1,I) = B(1,I) + 1.0
      DO J = 1,N-1
         A(J+1,I) = B(J+1,I) + 1.0
         C(J,I) = A(J+1,I) + 2.0
      ENDDO
      C(N,I) = A(N+1,I) + 2.0
    ENDDO
```

Alignment

- **Definition:** Let δ be a dependence between loops. The alignment threshold of δ is defined as follows:
  - —If δ is loop independent after merging, threshold(δ) = 0
  - —If δ is forward carried after merging, threshold(δ) is the negative of the resulting dependence threshold.
  - —If δ is fusion preventing, threshold(δ) is the threshold of the merged dependence.
- Aligning by the largest threshold allow fusion.

Mechanics

- After alignment, we're left with loops with mismatched bounds to be fused.
- We need a procedure for fusing these loops.

Weighted Fusion

- Different choices of which loops to fuse lead to different savings.
- We can annotate the loop-dependence graph with weights specifying cycles saved.
- We want to maximize total number of cycles saved

Weighted Fusion

- **Definition:** A mixed-directed graph is a graph $G = (V, E = E_d \cup E_u)$ where (V, E_d) forms a directed graph, (V, E_u) forms an undirected graph, and E_d and E_u are disjoint.
  - G is acyclic if (V,Ed) is acyclic.
  - w is a successor or predecessor if v if it is such in (V,Ed).
  - w is a neighbor of v if it is such in (V,Eu).

Weighted Fusion

- Definition: Let G be an acyclic mixed-directd graph, W a weight function on E, B a set of bad vertices, and Eb a set of bad edges. The weighted loop fusion problem is the problem of finding vertex sets {V1,V2,…,Vn} such that:
  - {V1,V2,…,Vn} partitions V.
  - Each vertex set Vi either contains no bad vertices, or consists of a single bad vertex.
  - Given two v and w in Vi, there is no path from v to w (in Ed) that leaves Vi.
  - Given v and w in Vi, there is no bad edge between v,w.
  - The induced graph on the vertex sets is acyclic.
  - $\sum_{i} \sum_{\substack{v,w \in V_i \\ (v,w) \in E}} W(v,w)$ is maximized.

Weighted Fusion

- The weighted-fusion problem is NP hard.
- As a result, we need to resort to heuristics.
- One heuristic that has proven highly effective, is the greedy heuristic.

Weighted Fusion

- **An efficient solution proposed by Kennedy:**
  - Initialize quantities, and compute successor, predecessor, and neighbor sets.
  - Topologically sort (V,Ed)
  - Compute the sets pathFrom(v) and badPathFrom(v)
  - Compute sets pathTo(v) and badPathTo(v)
  - Insert E into priority queue edgeHeap
  - While edgeHeap is non empty
    - Remove (v,w) from edgeHead
    - If w is in badPathFrom[v] then continue
    - Collapse v,w, and all edges on directed path between v and w into a node
    - Recompute all appropriate edges

Weighted Fusion

- If implemented properly, this algorithm runs in $O(EV + V^2)$ time.
- This algorithm does well in practice, but *generally* is not optimal.
- For multilevel loops, we fuse the outer loop and then recursively fuse inner loops.

Ordering Transformations

- **Recommended order:**
  - Loop Distribution
  - Loop Interchange
  - Loop alignment and fusion
  - Unroll-and-jam
  - Scalar Replacement

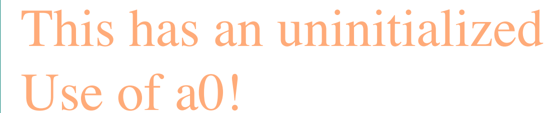
Loops With Conditionals

- Consider the following:

```
DO I = 1,M
      IF (M(I) .LT. 0) A(I) = B(I) + C
     D(I) = A(I) + E
   ENDDO
```

- Naïve replacement produces

```
DO I = 1,M
   IF (M(I) .LT. 0) THEN
     a0 = B(I) + C
     A(I) = a0
   ENDIF
   D(I) = a0 + E
ENDDO
```



Loops With conditionals

- This can be fixed by adding another branch to the conditional:

```
DO I = 1,M
   IF (M(I) .LT. 0) THEN
     a0 = B(I) + C
     A(I) = a0
   ELSE
     a0 = A(I)
   ENDIF
   D(I) = a0 + E
ENDDO
```

- More generally, we can use techniques from partial redundancy elimination to fix this.

Loops With Conditionals

- **Set up the following equations:**

$$alive_{out}(b) = \bigcap_{c \in succ(b)} alive_{in}(b)$$

$$alive_{in}(b) = used(b) \cup (alive_{out}(b) \setminus killed(b))$$

$$init_{in}(b) = \bigcap_{a \cup \in pred(b)} init_{out}(a)$$

$$init_{out}(b) = init_{in}(b) \cup assigned(b)$$

$$pinit_{in}(b) = \bigcup_{a \in pred(b)} pinit_{out}(a)$$

$$pinit_{out}(b) = pinit_{in}(b) \cup assigned(b)$$

$$insert_{in}(b) = used(b) \setminus pinit_{in}(b)$$

$$insert_{out}(b) = alive_{out}(b) \cap \neg pinit(b) \cap (\bigcup_{c \in succ(b)} pinit(b)$$

- **Insert assignment at each insert point.**

Trapezoidal Loops

Consider the following:

```
DO I = 2,99
   DO J = 1,I-1
      A(I,J) = A(I,I) + A(J,J)
   ENDDO
ENDDO
```

Unrolling we get:

```
DO I = 2,99,2
   DO J = 1,I-1
      A(I,J) = A(I,I) + A(J,J)
   ENDDO
   DO J = 1,I
      A(I+1,J) = A(I+1,I+1) + A(J,J)
   ENDDO
ENDDO
```

Jamming, we get:

```
DO I = 2,99,2
   DO J = 1,I-1
      A(I,J) = A(I,I) + A(J,J)
      A(I+1,J) = A(I+1,I+1) + A(J,J)
   ENDDO
   A(I+1,I) = A(I+1,I+1) + A(I,I)
ENDDO
```

Trapezoidal Loop

- Consider the following convolution code:

```
DO I = 0, N3
   DO J = I, MIN(N3, I+N2)
      F3(I) = F3(I) + F1(J)*W(I-J)
   ENDDO
   F3(I) = F3(I)*DT
ENDDO
```

- Partition into two parts:

```
DO I = 0, N3-N2
 DO J = I, I+N2
   F3(I) = F3(I) + F1(J)*W(I-J)
 ENDDO
 F3(I) = F3(I)*DT
ENDDO
```

```
DO I = N3-N2+1,N3
 DO J = I, N3
   F3(I) = F3(I) + F1(J)*W(I-J)
 ENDDO
 F3(I) = F3(I)*DT
ENDDO
```

No longer trapezoidal, so we can ignore it.

Trapezoidal Loops

Unroll-and-jam on the outer loop gives us:

```
DO I = 0,N3-N2,2
  F3(I) = F3(I) + F1(I)*W(0)
  DO J = I+1,I+N2
     F3(I) = F3(I) + F1(J)*W(I-J)
     F3(I+1) = F3(I+1) + F1(J)*W(I-J+1)
  ENDDO
  F3(I+1) = F3(I+1) + F1(I+N2+1)*W(-N2-1)
  F3(I) = F3(I)*DT
  F3(I+1) = f3(I+1)*DT
ENDDO
```

Trapezoidal Loops

- **Finally, with scalar replacement we have:**

```
DO I = 0,N3-N2,2
  f3I = F3(I); f3I1 = F3(I+1)
  f1I = F1(I); wIJ0 = W(0)
  f3I = f3I + f1I*wIJ0
  DO J = I+1,I+N2
    f1J = F1(J); wIJ1 = W(I-J)
    f3I = f3I + f1J*wIJ1
    f3I1 = f3I1 + f1J*wIJ0
    wIJ0 = wIJ1
  ENDDO
  f1J = F1(I+N2+1); wIJ1 = W(-N2-1)
  f3I1 = f3I1 + f1J*wIJ1
  F3(I) = f3I*DT; F3(I+1) = f3I1*DT
```

Experiments by Carr and Kennedy give a 2.22 speedup when run on a MIPS M230 with arrays of 100 elements.

Summary

- We've covered two general techniques:
  - —Loop Interchange -- creates opportunity for optimization
  - —Loop Fusion with Alignment -- brings uses together so they can share a register.
- We've also extended these techniques to loops with control flow and with trapezoidal iteration bounds.

Homework

- Reading list for next class
  - Chapter 9: Managing Cache
- Homework assignment for discussion in next class
  - Exercise 8.6