Program Analysis & Transformations

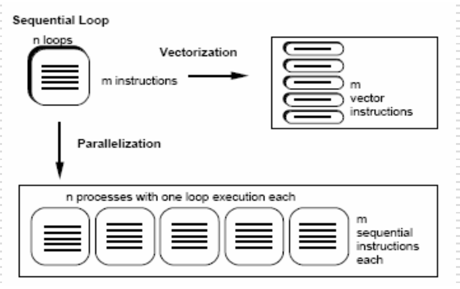
Loop Parallelization and Vectorization

Toheed Aslam

Background

- Vector processors
- Multi processors
- Vectorizing & parallelizing compilers
- SIMD & MIMD models

Loop Parallelism



Bräunl 2004

Data Dependence Analysis

- Flow dependence
- Anti dependence
- Output dependence

S_1 `x = …` S_2 `… = x`	S_1 `… = x` S_2 `x = …`	S_1 `x = …` S_2 `x = …`
$S_1\ \delta\ S_2$	$S_1\ \delta^{-1}\ S_2$	$S_1\ \delta^{o}\ S_2$

Vectorization

- Exploiting vector architecture

```
DO I = 1, 50
    A(I) = B(I) + C(I)
    D(I) = A(I) / 2.0
ENDDO
```

vectorize

```
A(1:50) = B(1:50) + C(1:50)
D(1:50) = A(1:50) / 2.0
```

Vectorization

```
A(1:50) = B(1:50) + C(1:50)
D(1:50) = A(1:50) / 2.0
```

```
vadd A[1], B[1], C[1], 50
vdiv D[1], A[1], SR, 50
```

```
mov VL, 50
vload V1, B[1], 1
vload V2, C[1], 1
vadd V1, V2, V3
vstore V3, A[1], 1
vdiv V3, SR, V4
vstore V4, D[1], 1
```

Vectorization

- Discovery
  - build data dependence graph
  - inspect dependence cycles
  - inspect each loop statement to see if target machine has vector instruction to execute accordingly
- Proper course of action?

Vectorization

- Transformation
  - loops with multiple statements must be transformed using the *loop distribution*
  - loops with no loop-carried dependence or has forward flow dependences

```
   DO I = 1, N
S1   A(I+1) = B(I) + C
S2   D(I) = A(I) + E
   ENDDO
```

loop distribution

```
   DO I = 1, N
S1   A(I+1) = B(I) + C
   ENDDO
   DO I = 1, N
S2   D(I) = A(I) + E
   ENDDO
```

vectorize

```
S1 A(2:N+1) = B(1:N) + C
S2 D(1:N) = A(1:N) + E
```

Vectorization

- Dependence Cycles
  - acyclic
    - solution: re-ordering of statements

```
   DO I = 1, N
S1   D(I) = A(I) + E
S2   A(I+1) = B(I) + C
   ENDDO
```

loop distribution

```
   DO I = 1, N
S2   A(I+1) = B(I) + C
   ENDDO
   DO I = 1, N
S1   D(I) = A(I) + E
   ENDDO
```

vectorize

```
S2 A(2:N+1) = B(1:N) + C
S1 D(1:N) = A(1:N) + E
```

Vectorization

- Dependence Cycles
  - cyclic
    - solution: statement substitution
    - otherwise, distribute loop
      - dependence cycle statements in a serial loop
      - rest of the loop as vectorized

```
   DO I = 1, N
S1   A(I) = B(I) + C(I)
S2   B(I+1) = A(I) + 2
   ENDDO
```

```
   DO I = 1, N
S1   A(I) = B(I) + C(I)
S2   B(I+1) = (B(I) + C(I)) + 2
   ENDDO
```

Vectorization

- Nested loops

```
   DO I = 1, N
S1   A(I, 1:N) = B(I, 1:N) + C(I, 1:N)
S2   B(I+1, 1:N) = A(I, 1:N) + 2
   ENDDO

S3     D(1:N, 1:N) = D(1:N, 1:N) + 1
```

- Conditions in loop

```
   DO I = 1, N
S1   IF( A(I) < 0)
S2     A(I) = -A(I)
   ENDDO
```

vectorize

```
WHERE(A(1:N) < 0) A(1:N) = -A(1:N)
```

Parallelization

- Exploiting multi-processors
- Allocate individual loop iterations to different processors
- Additional synchronization is required depending on data dependences

Parallelization

- ☐ Fork/Join parallelism
- ☐ Scheduling
  - Static
  - Self-scheduled

Parallelization

```
for i:=1 to n do
S1: A[i]:= C[i];
S2: B[i]:= A[i];
end;
```

- ☐ Data dependency: S1 δ(=) S2 (due to A[i])
- ☐ Synchronization required: NO

```
doacross i:=1 to n do
S1: A[i]:= C[i];
S2: B[i]:= A[i];
enddoacross;
```

Parallelization

- ☐ The inner loop is to be parallelized:

```
for i:=1 to n do
     for j:=1 to m do
     S1: A[i,j]:= C[i,j];
     S2: B[i,j]:= A[i-1,j-1];
     end;
end;
```

- ☐ Data dependency: S1 δ(<,<) S2 (due to A[i,j])
- ☐ Synchronization required: NO

```
for i:=1 to n do
     doacross j:=1 to m do
     S1: A[i,j]:= C[i,j];
     S2: B[i,j]:= A[i-1,j-1];
     enddoacross;
end;
```

Parallelization

```
for i:= 1 to n do
S1: A[i] := B[i] + C[i];
S2: D[i] := A[i] + E[i-1];
S3: E[i] := E[i] + 2 * B[i];
S4: F[i] := E[i] + 1;
end;
```

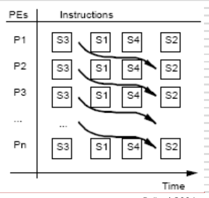
- ☐ Data Dependences:
  - S1 δ(=) S2 (due to A[i]) ← no synch. required
  - S3 δ(=) S4 (due to E[i]) ← no synch. required
  - S3 δ(<) S2 (due to E[i]) ← synch. required

Parallelization

- ☐ After re-ordering and adding sync code

```
var sync: array [1..r] of semaphore[0];

doacross i:=1 to n dc
 S3:  e[i] := e[i] + 2 * b[i];
      V(sync[i]);
 S1:  a[i] := b[i] + c[i];
 S4:  f[i] := e[i] + 1;
      if i>1 then P(sync[i-1]) end;
 S2:  d[i] := a[i] + e[i-1]
enddoacross;
```



Bräunl 2004

Review-I

- ☐ Data dependence within an instruction

```
for i:= 1 to n do
S1: A[i] := A[i+1]
end;
```

- ☐ Is this loop vectorizable?

Review-II

- Data dependence within an instruction

```
for j:= 1 to n do
S1: X[j+1] := X[j] + C
end;
```

- Is this loop vectorizable?

References

- *Optimizing Supercompilers for Supercomputers*, Michael Wolfe
- *Parallelizing and Vectorizing Compilers*, Rudolf Eigenmann and Jay Hoeflinger
- *Optimizing Compilers for Modern Architectures,* Randy Allen, Ken Kennedy