Enhancing Fine-grained parallelism

By -
Rajarshi Rakshit (05305024)
Dhananjay Muli (05305906)
Ashish Gudhe (05305028)

FINE-GRAINED PARALLELISM

- After Allen-Kennedy's codegen algo., what's next?
- Can we fine-grain the parallelism?
- Here we will look at some transformations that support it.

Loop Interchange

```
    DO I=1,N
         DO J=1,M
S:           A(I, J+1) = A(I, J) + B
         ENDDO
     ENDDO
```

- The transformed code after loop interchange can be vectorised

Loop Interchange

Here we interchange the loop I and loop J.

```
    DO I=1,N                                DO J=I,M
        DO J=1,M                                DO I=1,N
S:          A(I, J+1) = A(I, J) + B   -->  S:      A(I, J+1) = A(I, J) + B
        ENDDO                                   ENDDO
    ENDDO                                   ENDDO
```

- The transformed code after loop interchange can be vectorised

Loop Interchange

```
    DO I=1,N
        DO J=1,M
S:           A(I, J+1) = A(I, J) + B
        ENDDO
    ENDDO
```

```
    DO J=I,M
        DO I=1,N
S:           A(I, J+1) = A(I, J) + B
        ENDDO
    ENDDO
```

```
 DO J=I,M
S:   (1:N, J+1) = A(1:N, J) + B
 ENDDO
```

The transformed code after loop interchange, vectorised

Safety of Loop Interchange

Not all the loops can be interchanged

```
    DO I=1,M
        DO J=1,N
S:          A(I+1, J) = A(I, J+1) + B
        ENDDO
    ENDDO
```

```
    DO J=1,N
        DO I=1,M
S:          A(I+1, J) = A(I, J+1) + B
        ENDDO
    ENDDO
```

Safety of Loop Interchange(cont.)

```
   DO I=1,N
      DO J=1,M
         DO K=1,L
S:          A(I+1, J+1,K) = A(I, J,K) + A(I, J+1, K+1)
         ENDDO
      ENDDO
   ENDDO
```

– SOURCE :	A(I+1, J+1,K)	SOURCE :	A(I+1, J+1,K)
– SINK :	A(I, J,K)	SINK :	A(I, J+1,K+1)
– D.V. :	< , < , =	D.V. :	< , = , >

Safety of Loop Interchange(cont.)

```
 DO I=1,N
   DO J=1,N
      DO K=1,N
S :      A(I, J) = A(I, J) + B(I, K) + C(K, J)
      ENDDO
    ENDDO
  ENDDO
```

```
DO K=1,N
    DO I=1,N
        DO J=1,N
S:    A(I, J) = A(I, J) + B(I, K) + C
    (K, J)
        ENDDO
    ENDDO
ENDDO
```

Safety of Loop Interchange(cont.)

```
DO I=1,N
    DO J=1,N
        DO K=1,L
S:          A(I, J) = A(I, J) + B(I, K) + C(K, J)
        ENDDO
    ENDDO
ENDDO
```

```
DO I=1,N
    FORALL (J=1,N)
S:          A(1:N, J) = A(1:N, J) + B(1:N, K) + C(K, J)
    END FORALL
ENDDO
```

Scalar Expansion

Example - Swapping of two vectors:

```
      DO I =1, N
S1        T =  A(I)
S2        A(I) = B(I)
S3        B(I) = T
      ENDDO
```

Scalar Expansion

Code produced by scalar expansion :

```
        DO I =1, N
    S1        T$(I) =  A(I)
    S2        A(I) = B(I)
    S3        B(I) = T$(I)
        ENDDO
```

Vectorized code :

```
    S1        T$(1:N) =  A(1:N)
    S2        A(1:N) = B(1:N)
    S3        B(1:N) = T$(1:N)
```

Scalar Expansion

- Profitability of scalar expansion
- Deletable edges in the dependence graph
- Covering definition of a scalar

Scalar Expansion

- Edges that are deletable in the dependce graph by scalar expansion :
  - Backward carried antidependences
  - Forward carried output dependences
  - Loop independent antidependences into the covering definition
  - Loop carried true dependences from a covering definition

Scalar Renaming

Example :

```
     DO I = 1,100
S1      T = A(I) + B(I)
S2      C(I) = T + 2
S3      T = B(I) + 3 * A(I)
S4      B(I+1) = T – 4
     ENDDO
```

Scalar Renaming

Example :

```
    DO I = 1,100
S1      T = A(I) + B(I)
S2      C(I) = T + 2
S3      T = B(I) + 3 * A(I)
S4      B(I+1) = T – 4
    ENDDO
```

After scalar renaming :

```
    DO I = 1,100
S1      T1 = A(I) + B(I)
S2      C(I) = T1 + 2
S3      T2 = B(I) + D(I)
S4      C(I+1) = T2 – 4
    ENDDO
```

Array Renaming

- Similar to Scalar Renaming.
- Idea is to remove loop independent anti-dependence and output dependence.
- Identify a define-use pair and rename it.

```
DO I = 1 , N
S1 :  A[I] = A[I-1] + X
S2 :  Y[I] = A[I]  + Z
S3 :  A[I] = B[I] + C
ENDDO
```

```
DO I = 1 , N
S1 :  A$[I] = A[I-1]+X
S2 :  Y[I]   = A$[I] + Z
S3 :  A[I]   = B[I] + C
ENDDO
```

Vector code after array renaming :-

```
S3 : A(1:N)    =  B(1:N)    + C
S1 : A$(1:N)  =  A(0:N-1) + X
S2 : Y(1:N)    =  A$(1:N)  + Z
```

Index Set Splitting

- Dependence pattern in the iteration space.
- Idea is to split the loop based on the dependence.
- By splitting loop we are splitting the index of the array.

3 Methods of Index Set Splitting:-

- Threshold Analysis
- Loop peeling
- Section based splitting

Threshold Analysis

- Compute the threshold(dependence distance) for a reference pair.
- Breaks the loop into sizes smaller than the threshold.

Example for constant threshold.

```
DO I = 1 , 20
   A(I + 5) = A(I) + B
ENDDO
```

```
DO I = 1 , 20 , 5
    DO  J = I , I + 4
        A(J + 5) = A(J) + B
    ENDDO
ENDDO
```

Vector code generated :-

```
DO I = 1 , 20 , 5
    A(I+5, I+9) = A(I:I+4) + B
ENDDO
```

Threshold Analysis(cont)

Example for crossing threshold.

```
DO I = 1 , 100

     A[101 - I] = A[I] + B

ENDDO;
```

```
DO I = 1 , 100 , 50
  DO  J = I , I + 49
        A[101 - J] = A[J] + B
  ENDDO
ENDDO;
```

Vector code generated :-

```
DO I = 1 , 100 , 50
      A(101–I:51-I) = A(I:I+49) + B
 ENDDO
```

Loop Peeling

- Idea is to peel the loop or remove certain iterations outside the loop

Example which peels out a single iteration

```
DO J = 1 , N
  A(J) = A(J) + A(1)
ENDDO
```

→

```
A(1) = A(1) + A(1)
DO J = 2 , N
    A(J) = A(J) + A(1)
ENDDO
```

↓

Vector Code generated :-

```
A(1)     = A(1) + A(1)
A(2:N) = A(2:N) + A(1)
```

Loop Peeling(cont..)

Example where loop is split and loop carried dependency is converted into loop independent dependency.

```
DO I = 1 , N
   A(I) = A(N/2) + B(I)
ENDDO
```

→

```
M = N/2
DO I = 1 , M
      A(I) = A(N/2) + B(I)
ENDDO
DO I = M+1 , N
      A(I) = A(N/2) + B(I)
ENDDO
```

Section based splitting

- Variation of loop peeling where sections of loop involved in dependence are recognised.
- Loop is splitted to separate out the identified sections.

- Example :-

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

      DO J = 1 , N
      S2:  A(J,I+1) = B(J,I) + D
      ENDDO
ENDDO
```

Section based splitting(cont)

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

  DO  J = 1 , N/2
  S21:  A(J,I+1) = B(J,I) + D
  ENDDO

  DO  J = N/2+1 , N
  S22:   A(J,I+1) = B(J,I) + D
  ENDDO
ENDDO
```

Here S22 is independent of S1 and S12.

Distributing the I-loop we get :-

```
DO I = N/2+1 , N
  DO  J = N/2+1 , N
   S22: A(J,I+1) = B(J,I) + D
   ENDDO
ENDDO

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

   DO  J = 1 , N/2
   S21: A(J,I+1) = B(J,I) + D
   ENDDO
 ENDDO
```

Reference

Chapter 5 of

Optimizing Compilers for modern architectures : a dependence-based approach. By Ken Kennedy and John R. Allen.

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.