

VECTOR OPTIMIZATION TECHNIQUES

Notes prepared for EE 6481

by

Professor Cyrus D. Cantrell

May–August 2005

COMBINED VECTOR AND SCALAR OPERATIONS

- SAXPY (Scalar A times X Plus Y)

$$\mathbf{z} = \alpha\mathbf{x} + \mathbf{y}$$

- GAXPY (General A times X Plus Y)

$$\mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{y}$$

VECTORIZATION INHIBITORS

- Loops containing
 - ▷ Dependences involving an array
 - ▷ Recursion
 - ▷ I/O statements
 - ▷ Function or subroutine calls
 - ▷ References to a nonvectorizable statement function
 - ▷ A limit or stride that varies with respect to an outer loop
 - ▷ Multiple entrances or exits
 - ▷ Equivalenced variables
 - ▷ Computed or assigned GO TO statements
 - ▷ Vector reduction operations (on some vector computers)
 - ▷ Nonlinear array references to memory
 - ▷ Ambiguous subscripts

IMPORTANCE OF VECTOR LENGTH

- Suppose that the vector length is l_v , *i.e.* l_v vectors can be held in hardware registers.
- If we need s cycles to set up a vector pipeline, then we need $s + n$ cycles to perform n operations if $n \leq l_v$.
- If $n > l_v$, we need $n + s(\lfloor n/l_v \rfloor + 1)$ cycles to perform n operations. This reduces the rate of computation by the factor

$$L(n) = \frac{1}{1 + \frac{s}{n} (\lfloor n/l_v \rfloor + 1)}.$$

- The factor $n_{1/2}$, such that

$$L(n_{1/2}) = \frac{1}{2},$$

measures the startup overhead. A computer with a large value of $n_{1/2}$ performs poorly in calculations on short vectors.

VECTOR OPTIMIZATION TECHNIQUES (1)

- Design the algorithm to yield an optimized, vectorizable innermost DO loop:
 - ▷ Avoid nonvectorizable constructions
 - ▷ Avoid scalar temporary variables
 - ▷ Use unit-stride memory references
 - ▷ Unroll loops if there is only one pipe to memory
 - ▷ Distribute loops
 - ▷ Put longest vectors in innermost loop
 - ▷ Reorder loops or statements to avoid recurrence on innermost loop
- Use parallel operations or chaining to perform as much computation as possible in each vector period or cycle
- Use strip mining to minimize vector load overhead
- Minimize subroutine or function call overhead (several $\times 10^2$ CPU cycles per call)

VECTOR OPTIMIZATION TECHNIQUES (2)

- Avoid scalar temporary variables in a vector loop. Scalar temporaries force unnecessary scalar stores to memory, and may inhibit chaining on CRAYs.

▷ Slow matrix multiplication inner loop:

```
SUM = 0.  
DO 100 K = 1, N  
    SUM = SUM + A(J, K) * B(K, I)  
100 CONTINUE  
C(I, J) = SUM
```

▷ Faster loop:

```
C(I, J) = 0.  
DO 100 K = 1, N  
    C(I, J) = C(I, J) + A(J, K) * B(K, I)  
100 CONTINUE
```

▷ Some processors vectorize reduction operations such as the above dot product

VECTOR OPTIMIZATION TECHNIQUES (3)

- Use unit-stride memory references in the innermost loop to take maximum advantage of bank memory (if present), or if they will permit cache bypass:

- ▷ Non-unit-stride innermost loop:

```
DO 20 I = 1, M
  DO 20 J = 1, N
    A(I, J) = B(I, J) * C(I, J)
20 CONTINUE
  C(I, J) = SUM
```

- ▷ Interchange loops to obtain unit stride:

```
DO 20 J = 1, N
  DO 20 I = 1, M
    A(I, J) = B(I, J) * C(I, J)
20 CONTINUE
```

- ▷ Some compilers perform this optimization

VECTOR OPTIMIZATION TECHNIQUES (4)

- On a system with interleaved memory, avoid using strides through memory that have common factors with the number of memory banks

▷ Original loop:

```
DIMENSION A(32, N), B(32, N)
DO 100 I = 1, 32
  DO 100 J = 1, N
    A(I, J) = A(I, J) + B(I, J)
100 CONTINUE
```

- ▷ The inner loop has stride 32, which causes a bank conflict at every memory access on a machine with 32 banks (*e.g.*, CRAY X-MPs).

```
DIMENSION A(32, N), B(32, N)
DO 100 J = 1, N
  DO 100 I = 1, 32
    A(I, J) = A(I, J) + B(I, J)
100 CONTINUE
```

VECTOR OPTIMIZATION TECHNIQUES (5)

- Unroll an outer loop if the processor has only one vector pipeline between memory and the CPU (*e.g.*, the CRAY-1)
 - ▷ Unrolling can decrease conflicts between loads and stores
 - ▷ Dongarra and Eisenstat's SAXPY for $\mathbf{y} = \mathbf{Ax}$:

```
DO 20 J = 1, N
  DO 10 I = 1, M
    Y(I) = Y(I) + X(J) * A(I, J)
  10 CONTINUE
20 CONTINUE
```

- ▷ Each instance of the inner loop requires 2 vector loads, a (possibly chained) multiply-add, and a vector store, or a total of 3 memory references for 2 FLOPs

VECTOR OPTIMIZATION TECHNIQUES (6)

- Unroll the outer loop of Dongarra and Eisenstat's SAXPY to a depth of four:

```
DO 20 J = 1, N, 4
  DO 10 I = 1, M
    Y(I) = ((Y(I) + X(J-3) * M(I, J-3))
*          + X(J-2) * M(I, J-2))
*          + X(J-1) * M(I, J-1))
*          + X(J) * M(I, J)
  10 CONTINUE
20 CONTINUE
```

- Now each instance of the inner loop makes 6 memory references for 8 FLOPs, so that on a single-vector-pipe processor the speed is twice that of the original loop

VECTOR OPTIMIZATION TECHNIQUES (7)

- Distribute a DO loop for partial avoidance of recursion

▷ Original loop:

```
DO 20 I = 1, M
  A(I) = A(I-1) + B(I) * C(I)
20 CONTINUE
```

▷ Distributed loop:

```
DO 10 I = 1, M
  T(I) = B(I) * C(I)
10 CONTINUE
DO 20 I = 1, M
  A(I) = A(I-1) + T(I)
20 CONTINUE
```

▷ Some compilers perform such partial vectorizations automatically

VECTOR OPTIMIZATION TECHNIQUES (8)

- Distribute nested DO loops to enhance vectorization

▷ Original loops:

```
DO 20 I = 1, M
  DO 10 J = 1, N
    A(I) = A(I) + B(I, J) * C(I, J)
10  CONTINUE
    D(I) = E(I) + A(I)
20 CONTINUE
```

▷ Distributed and reordered loops:

```
DO 20 J = 1, N
  DO 10 I = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
10  CONTINUE
20 CONTINUE
DO 30 I = 1, M
  D(I) = E(I) + A(I)
30 CONTINUE
```

VECTOR OPTIMIZATION TECHNIQUES (9)

- Reorder loops to avoid multidimensional recursion

▷ Original loops:

```
DO 100 I = 1, M
  DO 100 J = 1, N
    A(I, J) = A(I, J-1) * B(I, J)
100 CONTINUE
```

▷ Reordered loops:

```
DO 100 J = 1, N
  DO 100 I = 1, M
    A(I, J) = A(I, J-1) * B(I, J)
100 CONTINUE
```

VECTOR OPTIMIZATION TECHNIQUES (10)

- Use **strip mining** to minimize vector load overhead

▷ Original loop:

```
DO 100 I = 1, M
  DO 100 J = 1, N
    A(I) = B(I) * C(I)
100 CONTINUE
```

▷ Strip-mined loop (assuming a processor with vector registers of length 128):

```
J = 0
DO 110 K = M, 0, -128
  DO 100 I = 1, 128
    A(I+J) = B(I+J) * C(I+J)
100 CONTINUE
  J = J + 128
110 CONTINUE
```

▷ If the upper limit M is not a multiple of 128, there will be some overhead in cleaning up the last part of the loop.

ARRAY DEPENDENCIES (1)

- Reference to a value calculated in a previous instance of the loop

▷ Nonvectorizable construction:

```
DO 100 I = 2, 100
  B(I) = A(I-1)
  A(I) = C(I)
100 CONTINUE
```

- ▷ If all of the elements of A, B and C were operated on in groups, then execution of $B(3) = A(2)$ would occur before execution of $A(2) = C(2)$.
- ▷ Vectorization is possible after interchanging the statements:

```
DO 100 I = 2, 100
  A(I) = C(I)
  B(I) = A(I-1)
100 CONTINUE
```

- ▷ Some vectorizing FORTRAN compilers make such statement interchanges automatically

ARRAY DEPENDENCIES (2)

- Reference to a value that should be calculated in a later instance of the loop

▷ Nonvectorizable construction:

```
DO 100 I = 1, 99
  A(I) = C(I)
  B(I) = A(I+1)
100 CONTINUE
```

▷ If all of the elements of A, B and C were operated on in groups, then execution of $A(2) = C(2)$ would occur before execution of $B(1) = A(2)$.

▷ Vectorization is possible after interchanging the statements:

```
DO 100 I = 2, 100
  B(I) = A(I+1)
  A(I) = C(I)
100 CONTINUE
```

▷ Some vectorizing FORTRAN compilers make such statement interchanges automatically