CSc 553

Principles of Compilation

32 : Scientific Codes

Department of Computer Science
University of Arizona

collberg@gmail.com

Scientific Programs

- In the next couple of lectures we will concentrate on *scientific programs*. These are programs used in science and engineering.
- As we will see, these programs don't look much like the programs you or I write. They don't manipulate objects, they don't use dynamic dispatch. Instead, they manipulate floating point arrays in (often) very regular patterns, using FOR-loops. And they're written in FORTRAN.
- The people who run these programs care deeply about speed. We will discuss two ways of speeding up scientific programs: parallelizing them and making efficient use of the memory hierarchy.

Who Needs Speed?

These are some of the traditional users/uses of high-performance computers:

weather forecasting

crypt- & image analysis Very secret!

aeronautical ind. Designing and testing aircrafts. Simulated wind tunnels. "Computational Fluid Dynamics" (CFD).

automotive ind. Simulated crash testing.

nuclear ind. Simulation of thermonuclear devices.

computer ind. Design and simulation of VLSI circuits.

pharmaceutical ind. Drug design.

What Do These Programs Look Like?

What Kind of Programs? I

- They are written i FORTRAN! Or sometimes C.
- They are often old, **DUSTY-DECK**, sequential, and difficult to maintain and rewrite.
- They use many (multidimensional) arrays.
- They consist largely of nested **FOR**-loops (called **DO**-loops in FORTRAN).
    - A **loop nest** is a set of loops one inside the next.
    - In a **perfect loop nest** every loop (except the innermost one) contains exactly one loop and nothing more.

What Kind of Programs? II

- A loop that accesses every fourth element of an array is **stride-4**, if it accesses every element (in order) it's **stride-1**, etc. The following loop has stride-3 accesses to `A`, and stride-6 access to `B`:

  ```
  FOR i := 1 to n BY 3 DO
     A[i] := B[2*i]
  END
  ```

- A number of benchmarks have been constructed to test how compilers/hardware handle these kinds of codes:
  1. Livermore loops (of Lawrence Livermore Labs)
  2. NAS benchmark
  3. Linpack

Common Operations

Scatter

```
FOR i := 1 TO n DO
   v[j[i]] := a[i]
END
```

Gather

```
FOR i := 1 TO n DO
   v[i] := a[j[i]]
END
```

Reductions

```
S := 0; P := 1;
FOR i := 1 TO n DO
   S := S + v[i]
   P := P * v[i]
END
```

Example Loops I

- Livermore loop: first sum.

FORTRAN

```fortran
      DO 11 k = 2,n
11      X(k)= X(k-1) + Y(k)
```

C

```c
for ( k=1 ; k<n ; k++ )
    x[k] = x[k-1] + y[k];
```

FORTRAN weirdness

- FORTRAN `DO`-loops:
  $\boxed{\texttt{DO foot dovar = inital,final,incr}}$. `foot` is a statement number (label). `incr` can be omitted.
- `CONTINUE` serves as a placeholder for a label. It does nothing.
- FORTRAN array references use `"A()"`, not `"A[]"`.
- Variables that start with `I,J,K` are always integers.
- `CONJG` is a built-in FORTRAN function that takes a complex number $x + iy$ (expressed as `(X, Y)` in FORTRAN) as argument and returns the **complex conjugate** $x - iy$. Just thought you'd like to know. . .
- In FORTRAN the comparison operators $<, \leq, =, \neq, >, \geq$ are called `.LT.`, `.LE.`, `.EQ.`, `.GT.`, `.GE.`, and `.NE.`.

Example Loops II

- Livermore loop: general linear recurrence equation.

FORTRAN

```
DO 6 i= 2,n
   DO 6 k= 1,i-1
      W(i)= W(i) + B(i,k) * W(i-k)
6 CONTINUE
```

C

```
for ( i=1 ; i<n ; i++ )
   for ( k=0 ; k<i ; k++ )
      w[i] += b[k][i] * w[(i-k)-1];
```

Example Loops III

- Livermore loop: matrix*matrix product

FORTRAN

```
DO 21 k= 1,25
   DO 21 i= 1,25
      DO 21 j= 1,n
         PX(i,j)= PX(i,j) + VY(i,k) * CX(k,j)
21 CONTINUE
```

C

```
for ( k=0 ; k<25 ; k++ )
   for ( i=0 ; i<25 ; i++ )
      for ( j=0 ; j<n ; j++ )
         px[j][i] += vy[k][i] * cx[j][k];
```

Example Loops IV

- Linpack: constant (da) times a vector (dx incremented by incx) plus a vector (dy incremented by incy).

```
daxpy(n,da,dx,incx,dy,incy)
  double dx[],dy[],da; int incx,incy,n; {
  int i,ix,iy,m,mp1;
  if ((n <= 0) || (da == 0.0)) return;
  if(incx != 1 || incy != 1) {
    ix = 0; iy = 0;
    if(incx < 0) ix = (-n+1)*incx;
    if(incy < 0)iy = (-n+1)*incy;
    for (i = 0;i < n; i++) {
      dy[iy] = dy[iy] + da*dx[ix];
      ix = ix + incx; iy = iy + incy;}
    return; }
  for (i = 0;i < n; i++) dy[i] = dy[i] + da*dx[i];}
```

Example Loops V (a)

- NAS Benchmark: complex radix 2 ffts on the first dimension of the 2-d array x.

```
SUBROUTINE CFFT2D1 (IS,M,M1,N,X,W,IP)
   COMPLEX X(M1,N), W(M), CT, CX
   INTEGER IP(2,M)
   DATA PI/3.141592653589793/
 DO 110 I = 1, M
   IP(1,I) = I
110 CONTINUE
   L =I1 = 1
```

Example Loops V (b)

```
120 I2 = 3 - I1
    DO 130 J = L, M2, L
       CX = W(J-L+1)
       IF (IS .LT. 0) CX = CONJG (CX)
       DO 130 I = J-L+1, J
          II = IP(I1,I)
          IP(I2,I+J-L) = II
          IM = IP(I1,I+M2)
          IP(I2,I+J) = IM
          DO 130 K = 1, N
             CT = X(II,K) - X(IM,K)
             X(II,K) = X(II,K) + X(IM,K)
             X(IM,K) = CT * CX
130    CONTINUE
```

Example Loops V (c)

```
      L = 2 * L
      I1 = I2
      IF (L .LE. M2) GOTO 120

      DO 150 I = 1, M
         II = IP(I1,I)
         IF (II .GT. I) THEN
         DO 140 K = 1, N
            CT = X(I,K)
            X(I,K) = X(II,K)
            X(II,K) = CT
140      CONTINUE
   ENDIF
150   CONTINUE
   RETURN
END
```

Floating Point Computations

Floating Point Computation I

- All scientific programs manipulate floating point numbers. Many transformations that are legal on an integer expression are unsafe on the equivalent floating point expressions.
- In IEEE floating point
$$x * 0 = 0$$
may not be true, since if $x = \infty$,
$$\infty * 0 = NaN$$
($NaN \equiv Not\ a\ Number$).
- Similarly,
$$x + 0 = 0$$
may not be true, since if $x = NaN$, $x + 0$ would generate an exception, but the right hand side wouldn't.

Floating Point Computation II

- Let $\mathcal{R}_\infty$ be the largest FP number. Then

$$1.0 + (\mathcal{R}_\infty - \mathcal{R}_\infty) = 1.0$$

  but

$$(1.0 + \mathcal{R}_\infty) - \mathcal{R}_\infty = 0.0$$

- We can often safely convert a division by a constant into the equivalent multiplication. For example,

$$X/16.0$$

  can be transformed to

$$X * 0.0625$$

  because both 16.0 and 0.0625 can be represented exactly.

Floating Point Computation III

- Scientific programs iterate over arrays of floating point numbers. We will often want to transform these (for-) loops to improve efficiency. We still have to be careful to maintain correctness.
- Assume that we want to sum the elements of the following array:

[1]	[2]	[3]	[4]	[5]	[6]	[7]	$[\cdots]$	$[3*n]$
1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\cdots$	$-\mathcal{R}_\infty$

- In the next few slides we'll show how the access pattern will affect the result of the summation.

Floating Point Computation IV

- Accessing the array in stride-1 yields a result of $\boxed{0.0}$.

```
s := 0.0; n := 3
FOR i := 1 TO 3*n DO
   s := s + A[i]
ENDFOR
```

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$
$\uparrow 1$	$\uparrow 2$	$\uparrow 3$	$\uparrow 4$	$\uparrow 5$	$\uparrow 6$	$\uparrow 7$	$\uparrow 8$	$\uparrow 9$

$$((((((((1.0+\mathcal{R}_\infty)-\mathcal{R}_\infty)+1.0)+\mathcal{R}_\infty)-\mathcal{R}_\infty)+1.0)+\mathcal{R}_\infty)-\mathcal{R}_\infty = 0.0$$

Floating Point Computation V

- Accessing the array backwards yields the result of $\boxed{1.0}$.

```
s := 0.0; n := 3
FOR i := 3*n TO 1 BY -1 DO
   s := s + A[i]
ENDFOR
```

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$
$\uparrow 9$	$\uparrow 8$	$\uparrow 7$	$\uparrow 6$	$\uparrow 5$	$\uparrow 4$	$\uparrow 3$	$\uparrow 2$	$\uparrow 1$

$$(((((((-\mathcal{R}_\infty+\mathcal{R}_\infty)+1.0)-\mathcal{R}_\infty)+\mathcal{R}_\infty)+1.0)-\mathcal{R}_\infty)+\mathcal{R}_\infty)+1.0 = 1.0$$

Floating Point Computation VI

- Accessing the array backwards in blocks of 3 yields $\boxed{n}$.

```
s := 0.0; n := 3
FOR i := 3*n TO 3 BY -3 DO
   t := 0.0
   FOR j := i TO i-2 BY -1 DO t := t + A[j] ENDFOR
   s := s + t
ENDFOR
```

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$	1.0	$\mathcal{R}_\infty$	$-\mathcal{R}_\infty$
$\uparrow 9$	$\uparrow 8$	$\uparrow 7$	$\uparrow 6$	$\uparrow 5$	$\uparrow 4$	$\uparrow 3$	$\uparrow 2$	$\uparrow 1$

$$((-\mathcal{R}_\infty + \mathcal{R}_\infty) + 1) + \cdots + ((-\mathcal{R}_\infty + \mathcal{R}_\infty) + 1) = 3.0$$

Readings and References

- David Goldberg, *What Every Computer Scientist Should Know about Floating-Point Arithmetic.*, ACM Computing Surveys, Volume 23, Number 1, 1991, http://www.acm.org/pubs/citations/journals/surveys/1991-23-1/p5-goldberg/.