Computer Application for Scientific Computation
Assignment#01

2015-17231
박우정

1-1. Perform matrix matrix multiplication by changing Loop order as follows.

Matrix matrix multiplication의 속도를 측정하기 위해 아래의 Fortran90 구문을 이용하자.

```
program Loop_change
implicit none
integer, parameter::n=1024
integer::i,j,k,count1,count2,cr,cmax
real, dimension(n,n)::A,B,C
real::time
A=0.0;B=0.0
do i=2,n-1
   A(i,i-1)=1.0;A(i,i)=2;A(i,i+1)=1.0;
   B(i,i-1)=2.0;B(i,i)=-4.0;B(i,i+1)=2.0;
enddo
A(1,1)=2.0;A(1,2)=1.0;A(n,n-1)=1.0;A(n,n)=2.0
B(1,1)=-4.0;B(1,2)=2.0;B(n,n-1)=2.0;B(n,n)=-4.0
C=0.0
call system_clock(count1, cr, cmax)
do i=1,n
   do j=1,n
      do k=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
call system_clock(count2,cr,cmax)
time=dble(count2-count1)/dble(cr)
print*, 'naive time=', time, ' seconds'
end program Loop_change
```

이제 연산순서에 따른 속도를 비교하기 위해 i, j, k의 순서를 바꾸어가며 측정해보자. 일단 다음과 같이 6가지를 생각해볼 수 있다.

$$(i,j,k), (i,k,j), (j,i,k), (j,k,i), (k,i,j), (k,j,i)$$

1-2. Report the performance and analysis the results.

이 순서대로 위 프로그램을 실행해보면 다음과 같은 결과를 얻을 수 있다.

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   9.30500031      seconds
```

(i, j, k)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   11.1379995      seconds
```

(i, k, j)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   9.22299957      seconds
```

(j, i, k)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.51999998      seconds
```

(j, k, i)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   11.1580000      seconds
```

(k, i, j)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.69399977      seconds
```

(k, j, i)

따라서 속도가 빠른 순서대로 아래와 같은 비교표를 작성할 수 있다.

Loop Order	Time(sec)
(j, k, i)	5.51999998
(k, j, i)	5.69399977
(j, i, k)	9.22299957
(i, j, k)	9.30500031
(i, k, j)	11.1379995
(k, i, j)	11.1580000

즉, (1,1), (2,1), (3,1), ... , (n, 1) 순서로 column의 방향을 따라가면서 계산하는 것이 제일 빠른데, 실제로 Fortran90이 해당 계산을 columnwise processing하는 사실을 실험적으로 확인할 수 있었다.

다음은 더욱 유의미한 차이를 확인해보기 위해 $n = 2048$이라 두고, 똑같이 실행해본 결과다.

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   67.8669968      seconds
```

(i, j, k)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   85.8150024      seconds
```

(i, k, j)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   67.2839966      seconds
```

(j, i, k)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   40.0699997      seconds
```

(j, k, i)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   86.5479965      seconds
```

(k, i, j)

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   40.2879982      seconds
```

(k, j, i)

여기서도 속도가 빠른 순서대로 정리하면 다음과 같은 표를 얻을 수 있다.

Loop Order	Time(sec)
(j, k, i)	40.0699997
(k, j, i)	40.2879982
(j, i, k)	67.2839966
(i, j, k)	67.8669968
(i, k, j)	85.8150024
(k, i, j)	86.5479965

속도의 순서는 $n = 1024$일 때와 같고, 세 번째 원소, 즉 어떤 방향을 우선하여 계산할 지를 정해주는 것이 속도에 가장 큰 영향을 준다는 사실까지 알 수 있다. 물론, column, product, row 순서로 빠르다.

2. Apply the Loop Unrolling for the matrix matrix multiplication. Try possible unrolling for i,j,k loops and report the performance and analyze.

Loop unrolling for the matrix matrix multiplication을 알아보기 위해 다음과 같은 포트란 구문을 이용하자. 1에서 (j,k,i)가 제일 빠른 것을 알았으므로, 순서는 (j,k,i)로 유지하되, 각 do구문을 unroll하는 구문을 작성해보면 된다. 다음은 i에 대하여 4-steps loop unrolling 하는 프로그램이다.

```
program Loop_unrolled
implicit none
integer, parameter::n=1024
integer::i,j,k,count1,count2,cr,cmax
real, dimension(n,n)::A,B,C
real::time
A=0.0;B=0.0
do i=2,n-1
    A(i,i-1)=1.0;A(i,i)=2;A(i,i+1)=1.0;
    B(i,i-1)=2.0;B(i,i)=-4.0;B(i,i+1)=2.0;
enddo
A(1,1)=2.0;A(1,2)=1.0;A(n,n-1)=1.0;A(n,n)=2.0
B(1,1)=-4.0;B(1,2)=2.0;B(n,n-1)=2.0;B(n,n)=-4.0
C=0.0
call system_clock(count1, cr, cmax)
do j=1,n
   do k=1,n
      do i=1,n-3,4
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
         C(i+1,j)=C(i+1,j)+A(i+1,k)*B(k,j)
         C(i+2,j)=C(i+2,j)+A(i+2,k)*B(k,j)
         C(i+3,j)=C(i+3,j)+A(i+3,k)*B(k,j)
      end do
   end do
end do
call system_clock(count2,cr,cmax)
time=dble(count2-count1)/dble(cr)
print*, 'naive time=', time, ' seconds'
end program Loop_unrolled
```

unroll하는 loop를 조정하기 위해서는 I에 해당하는 do 구문의 끝을 줄이고, 간격을 늘린 후, 위처럼 행을 추가하면 된다.

j나 k Loop을 Unrolling하는 구문은 진하게 칠한 부분을 다음과 같이 바꾸면 된다. 다음 구문들은 모두 4-steps 기준으로 작성되었다.

$(i)\ j-th$ unrolling

```
do j=1,n-3,4
   do k=1,n
      do i=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
         C(i,j+1)=C(i,j+1)+A(i,k)*B(k,j+1)
         C(i,j+2)=C(i,j+2)+A(i,k)*B(k,j+2)
         C(i,j+3)=C(i,j+3)+A(i,k)*B(k,j+3)
      end do
   end do
end do
```

$(ii)\ k-th$ unrolling

```
do j=1,n
   do k=1,n-3,4
      do i=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)+A(i,k+1)*B(k+1,j)+A(i,k+2)*B(k+2,j)+A(i,k+3)*B(k+3,j)
      end do
   end do
end do
```

이제 각각 unrolled되는 문자에 대하여 2-step부터 4-step까지 속도를 각각 비교해보자. 각각 구문을 수정해가며 프로그램을 실행시키면 다음과 같은 결과를 얻을 수 있다.

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.19299984      seconds
```

$i-2\,steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.17600012      seconds
```

$i-3steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   4.96999979      seconds
```

$i-4steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.41400003      seconds
```

$j-2\,steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.66300011      seconds
```

$j-3steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.34000015      seconds
```

$j-4steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   5.45699978      seconds
```

$k-2steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   3.74300003      seconds
```

$k-3steps$

```
[a2015-17231@workstation1 hw1]$ ./a.out
 naive time=   3.55999994      seconds
```

$k-4steps$

이 결과들을 정리하면 다음과 같은 표를 얻을 수 있다.

order \ step	2	3	4
i	5.19299984	5.17600012	4.96999979
j	5.41400003	5.66300011	5.34000015
k	5.45699978	3.74300003	3.55999994

step의 수를 늘리면 대체적으로 빨라지는 것을 확인할 수 있다. 따라서 (기계가 허용하는 한) 많은 loop unrolling을 시행하면 순서를 반드시 지킬 필요 없이, 동시에 연산 processing이 가능하므로 빠른 연산을 할 수 있다.
한편, 위 구문에서 k는 연산 시 2개의 주소를 찾아 할당해야 하므로, loop unrolling의 효과를 더 크게 볼 수 있다.