AMath 483/583 — Lecture 10 — April 18, 2011

Today:

- Cache considerations
- Optimizing Fortran codes
- Debugging Fortran

Wednesday:

- Software packages
- LAPACK and BLAS

Read: Class notes and references
There are several new sections!

Notes:

Code optimization

We will look at some basics worth keeping in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary.
  "Premature optimization is the root of all evil" (Don Knuth)
- If so, determine which parts of code need to be improved and spend effort on these sections.
- Use optimized software such as BLAS, LAPACK.

Notes:

Memory Hierachy

Between registers and memory there are 2 or 3 levels of cache, each larger but slower.

Registers: access time 1 cycle

L1 cache: a few cycles

L2 cache: $\sim$ 10 cycles

(Main) Memory: $\sim$ 250 cycles

Hard drive: 1000s of cycles

Notes:

Array ordering — which loop is faster?

```
integer, parameter :: m = 4097, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
       a(i,j) = 0.d0
       enddo
    enddo

do j = 1,n
    do i=1,m
       a(i,j) = 0.d0
       enddo
    enddo
```

First: 0.72 seconds, Second: 0.19 seconds

Notes:

Much worse if m is high power of 2

```
integer, parameter :: m = 4096, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
       a(i,j) = 0.d0
       enddo
    enddo

do j = 1,n
    do i=1,m
       a(i,j) = 0.d0
       enddo
    enddo
```

First: 2.4 seconds, Second: 0.19 seconds

Notes:

More about cache

Simplied model of one level direct mapped cache.

32-bit memory address: 4.3×10^9 addresses

Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.
9 low order bits of memory address determine cache address.

For a memory fetch:

- Determine cache address, check if this holds desired words from memory.
- If so, use it.
- If not, check "dirty bit" to see if has been modified since load.
- If so, write to memory before loading new cache line.

Notes:

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.

Cache line holds 64 bytes = 8 floats. $4096/8 = 512$ cache lines per column of matrix.

Loading one column of matrix will fill up cache lines $0, 1, 2, \ldots, 511$.

Second column will go back to cache line 0.
But all elements in cache have been used before this happens,
Prefetching can be done by optimizing compiler.

Worse — Going across the rows:

The first 8 elements of column 1 go to cache line 0.

The first 8 elements of column 2 also map to cache line 0.

Similarly for all columns. The rest of cache stays empty.

Notes:

More about cache

If cache holds more lines:

1024 lines $\Longrightarrow$
first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

In practice cache is often set associative: small number of cache addresses for each memory address.

Notes:

Padding

Matrix dimensions that are high powers of 2 should usually be avoided.

Even though natural for some algorithms such as FFTs

May be worth declaring larger arrays and only using part of it.

Notes:

Matrix transpose

```
do j=1,n
    do i=1,n
        b(j,i) = a(i,j)
        enddo
    enddo
```

Accessing a by column but b by row!

Better to do by blocks — illustrate on board.

See also: **Bill Gropp's class at Illinois**, Lecture 2

Notes:

Matrix transpose

Suppose stride s divides n. Then can rewrite as:

Strip mining:

```
do jj=1,n,s
    do j=jj,jj+s-1
        do ii=1,n,s
            do i=ii,ii+s-1
                b(j,i) = a(i,j)
```

Loop reordering:

```
do jj=1,n,s
    do ii=1,n,s
        do j=jj,jj+s-1
            do i=ii,ii+s-1
                b(j,i) = a(i,j)
```

Loops over blocks in outer loops, within block in inner loops.

Notes:

Block matrix multiply

Compute $C = AB$. Can partition into blocks:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

When blocks A_{11} and B_{11} are in cache can compute the $A_{11}B_{11}$ *part* of $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Might next bring in B_{12} and compute the $A_{11}B_{12}$ *part* of $C_{12} = A_{11}B_{12} + A_{12}B_{22}$

Notes:

Flop rate for matrix multiply/add

a, b each 1000×1000 matrices.

Compare time of `c = matmul(a,b)` vs. `c = a+b`.

Compare megaflops per second: `1e-6*nflops/(t2-t1)`.

```
Add: CPU time (sec):  0.00687200
 rate:    145.52 megaflops/sec

Multiply: CPU time (sec):  2.38393500   slower
 rate:    838.53 megaflops/sec          higher
```

For addition: `nflops = n**2`
For multiplication: `nflops = (2n-1)*n**2`,
More flops, but each element is used n times,
$\Longrightarrow$ More flops per memory access $\Longrightarrow$ higher rate.

Notes:

Optimizing Fortran

See the examples at

`$CLASSHG/codes/fortran/optimize`.

`$CLASSHG/codes/particles`.

Notes:

Developing programs to minimize bugs

- Start simple and add features slowly
  Tackle stripped-down version of problem first
- Modularize: break problem into pieces
  Subroutines or functions with
  well-defined inputs and outputs
  Develop and debug separately first

Notes:

Developing programs to minimize bugs

Unit tests: Test small pieces (early and often)

- Python has a unittest module to assist,
- Allows specification of test cases, test suites.

Regression testing:

Test that adding a new feature (or fixing a bug) didn't break old features.

Keep sample programs that test various features of the code, Run these after making improvements or "fixing" a bug.

Notes:

Debugging in Fortran

Need to compile with `-g` flag, no optimization.

(Runs slower, so recompile once debugged.)

gdb — command line debugger similar to pdb.

ddd — GUI front end for gdb, can be obtained on VM via:

```
$ sudo apt-get install ddd
```

Eclipse — IDE that uses gdb.

Much better commercial debuggers available, e.g. totalview.

Notes:

Debugging Fortran

See the examples at

`$CLASSHG/codes/fortran/debug.`

Notes:

Segmentation faults

Sometimes running a program gives:

```
$ ./a.out
Segmentation Fault
```

This generally means the code tried to write to a part of memory where it didn't have permission.

Or:

```
$ ./a.out
Bus error
```

This generally means a bad address not even in memory.

Often these are a result of an array index out of bounds.

Notes:

Segmentation faults

```
integer :: i
real(kind=8), dimension(10) :: x

do i=1,15
    x(i) = 20.d0
    print *, "i = ",i
    print *, x(i)
    enddo
```

produces:

```
 ...
 i =           10
   20.0000000000000
 i =   1077149696
Segmentation fault
```

Why? `x(11) points to memory where i is stored!`

Notes:

Overwriting variables

```
integer :: i
real(kind=8), dimension(10) :: x

do i=1,15
    x(i) = 0.d0
    print *, "i = ",i
    print *, x(i)
    enddo
```

Goes into an infinite loop — `i` gets reset to 0.

Notes:

Array bounds checking

```
$ gfortran -fbounds-check run1.f90
```

Gives:

```
...
 i =           10
   20.0000000000000
Fortran runtime error: Array reference out of bound
for array 'x', upper bound of dimension 1 exceeded
(in file 'demo1.f90', at line 11)
```


Notes: