AMath 483/583 — Lecture 9 — April 15, 2011

Today:

- Computer architecture
- Cache considerations

Monday:

- Optimizing Fortran codes
- Debugging Fortran

Read: Class notes and references

How fast are computers?

Kilo = thousand (10^3)
Mega = million (10^6)
Giga = billion (10^9)
Tera = trillion (10^{12})
Peta = 10^{15}
Exa = 10^{18}

Processor speeds usually measured in Gigahertz these days.

Hertz means "machine cycles per second".

One operation may take a few cycles.

So a 1 GHz processor (10^9 cycles per second) can do
$> 100,000,000$ floating point operations per second
(> 100 Megaflops).

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C.

Each element c_{ij} is the inner product of
ith row of A with jth column of B.

Requires n multiplications and $n - 1$ additions to compute c_{ij}.

n^2 elements in $C \implies$ Requires $\mathcal{O}(n^3)$ floating point ops total.
Note: $n = 10,000 \implies n^3 = 10^{12}$ (≈ 1000 seconds)

These days, the bottle neck is often
getting data to and from the processor!

Note that each element of A, B is used n times.

Memory Hierachy

(Main) Memory: "Fast" memory that is hopefully large enough to contain all the programs and data currently running.

(But not nearly fast enough to keep up with CPU.)

Typically 1 – 4 GB.

Recall GB = gigabyte = 10^9 bytes = 8×10^9 bits.

For example, 1GB holds a single $10,000 \times 10,000$ matrix of floating point values (8 bytes each),
or 125 matrices that are each 1000×1000.

Hard Drive: Slower memory that contains data (including photos, video, music, etc.) and all programs you might want to use.

Typically 80 – 500 GB. (Slower but cheaper.)

CPU and registers

CPU — central processor unit

Executes instructions such as add or multiply.

Takes data from registers, performs operations, stores back to registers.

Transferring between registers and processor is very fast.

Different types of registers, e.g.

- Integer, floating point
- instruction registers
- address registers

Generally a very small number of registers.

Data and instructions must be transferred between other memory and registers as needed.

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

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Exploit parallelism to hide latency and increase throughput.

Even a "single core" machine has lots of things going on at once.

For example:

- Pipelined operations
- Executing / fetching / storing
- Prefetching future instructions
- Prefetching data into cache

5-stage instruction pipeline for RISC machine

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

IF = Instruction Fetch,
ID = Instruction Decode,
EX = Execute,
MEM = Memory access,
WB = Register write back.

http://en.wikipedia.org/wiki/Instruction_pipeline

Reducing memory latency

Reduce memory fetches by reusing data in cache as much as possible. Requires temporal locality.

Very simple Python example: if `len(x)` much larger than cache size,

```
z = 0.; w = 0.
for i in range(len(x)):
    z = z + x[i]
for i in range(len(x)):
    w = w + 3. * x[i]
```

should be rewritten as

```
for i in range(len(x)):
    z = z + x[i]
    w = w + 3. * x[i]
```

Note: Both are bad in Python, use e.g. `w = np.sum(3.*x)`

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.

A cache line typically holds 64 or 128 consecutive bytes (8 or 16 floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Another cache line will be brought from higher up the hierachy, and may displace some variables in cache.

Those cache lines will first have to be written back to memory.

Bottom line: Good to do lots of work on each set of data while in cache, before it has to be written back.

Organize algorithm for Temporal locality.

Spatial locality

Also good to organize algorithm so data that is consecutive in memory is used together when possible.

If data you need is scattered through memory, many cache lines will be needed and will contain data you don't need.

This is called spatial locality.

Matrix storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[1,2,3], [4,5,6]]
```

Or, could be stored by columns, as done in Fortran!

Matrix storage

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

```
loc 3401    Apy[0,0] = 10      Afort(1,1) = 10
loc 3402    Apy[0,1] = 20      Afort(2,1) = 40
loc 3403    Apy[0,2] = 30      Afort(1,2) = 20
loc 3404    Apy[1,0] = 40      Afort(2,2) = 50
loc 3405    Apy[1,1] = 50      Afort(1,3) = 30
loc 3406    Apy[1,2] = 60      Afort(2,3) = 60
```

Note: integer arithmetic must be done to convert (i, j) to memory address: If matrix is $m \times n$,

loc-py $= 3401 + in + j$, loc-fort $= 3400 + (j-1)m + i$

Aside on np.flatten

The np.flatten method converts an N-dim array to a 1-dimensional one:

```
>>> A = np.array([[10.,20,30],[40,50,60]])
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])


>>> A.flatten()    # Default is 'C'
array([ 10.,  20.,  30.,  40.,  50.,  60.])


>>> A.flatten('F') # Fortran ordering
array([ 10.,  40.,  20.,  50.,  30.,  60.])
```


Aside on np.reshape

The np.reshape method can go through data in either order:

```
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])

>>> A.reshape(3,2)    # order='C' by default
array([[ 10.,  20.],
       [ 30.,  40.],
       [ 50.,  60.]])

>>> A.reshape((3,2),order='F')
array([[ 10.,  50.],
       [ 40.,  30.],
       [ 20.,  60.]])
```


Spatial locality

Also good to organize algorithm so data that is consecutive in memory is used together when possible.

If data you need is scattered through memory, many cache lines will be needed and will contain data you don't need.

This is called spatial locality.

Spatial locality

Suppose A is $n \times n$ matrix,
D is $n \times n$ diagonal matrix with diagonal elements d_i.

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Python?? Same number of flops!

```
for i in range(n):
    for j in range(n):
        b[i,j] = d[i] * a[i,j]
```

or

```
for j in range(n):
    for i in range(n):
        b[i,j] = d[i] * a[i,j]
```

Answer: First one faster in Python (but loops still slow!)

Spatial locality

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Fortran?? Same number of flops!

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

or

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

Answer: Second one faster in Fortran!

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

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

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.

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.

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

Second column will go back to cache line 0 (same cache address).

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 is empty.

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.