Profiling/Optimization

Do You Want Your Code to Run Fast?

- hopefully yes!
- Algorithms can make a huge difference - FFT vs. (slow) DFT.
- Even with fast algorithm, you still win if you code it well.

What Takes Time?

- It’s very useful to have an idea of how long things take.
- Floating point - AVX512 e.g. lets CPU crunch on 512 bits at a time. For 64 bit numbers, this is 8 operations per cycle. Often FMAD (fused multiply-add) does 1 multiply and 1 add in one cycle. 4 cores*8 FMADS/cycle*2 ops/FMAD*3 GHz = 200 GFLOPS.
- This is a hard limit - we’ll never go faster than this.

```python
nn=[100,300,1000,3000,10000]
for n in nn:
    x=np.random.randn(n,n)
    t1=time.time()
    y=np.dot(x,x)
    t2=time.time()
    nops=2*n**3
    gflops=nops/(t2-t1)/1e9
    print("For matrix size " + repr(n) + " we have " + repr(gflops) + " GFLOPS.")
```

```
[Jonathans-MacBook-Pro:performance sievers$ python time_matrix_multiply.py
 For matrix size 100 we have 10.143419588875455 GFLOPS.
 For matrix size 300 we have 57.26736182048041 GFLOPS.
 For matrix size 1000 we have 108.21915758240341 GFLOPS.
 For matrix size 3000 we have 147.48941557071524 GFLOPS.
 For matrix size 10000 we have 155.02978165356996 GFLOPS.
```

```
Processes: 505 total, 4 running, 501 sleeping, 2464 threads
Load Avg: 2.31, 1.83, 1.68  CPU usage: 24.64% user, 2.75% sys, 72.59% idle  SharedLibs: 248M resident, 6
PhysMem: 31G used (4913M wired), 840M unused. VM: 2899G vsize, 1369M framework vsize, 23770976(0) swapin
Disks: 11135706/552G read, 16884444/417G written.

PID    COMMAND      %CPU  TIME     #TH    #WQ  #PORTS MEM    PURG  CMPRS PGRP  PPID STATE    BOOSTS
37286  Python       299.9 00:11.58 3/3    0    23     1633M  0B    0B    37286 498  running  *0[1]
25139  steam_osx    100.1 04:25:19 32/1   3    1541   352M   680K  137M  25139 1    running   0[6485]
138    WindowServer 11.1  13:44:50 11     4    6544   1408M  43M+  131M- 138   1    sleeping *0[1]
37285  top          6.1   00:01.41 1/1    0    28     10M    0B    0B    37285 442  running  *0[1]
0      kernel_task  3.8   05:51:10 226/16 0    0      679M   0B    0B    0     0    running   0[0]
```

Only using 3 cores, so expect ~150 GFLOPS.
Matrix multiplies can be very efficient!

Memory Bandwith

- Another bottleneck - data has to get to and from memory to the CPU.
- A few 10s of GBs is optimistic here.
- How many numbers per second can I get to my CPU(s)?
- How many operations do I need to do while they're there to be floating-point limited?
- Say 50 GB/s, 16 bytes (2 numbers), 3 billion #'s/s. At 192 GFLOPS, need to do 64 floating point operations.

Matrix Multiply

- If I wrote matrix multipy as:

- for i in range(n):
  for j in range(m):
    for ii in range(k):
      c[i,j]=c[i,j]+a[i,ii]*b[ii,j]

- Every element of a&b gets pulled through i times. For 10k by 10k, this is $2*10000^3*8$ bytes = 16TB. At 50 GB/s, this takes 16*(1000/50)=320s. 30x longer than it actually took.

Moral of the Story (so far)

- When you write code, think about operations *and* bandwidth.
- Your code will almost certainly be bandwidth-limited.

Profiling

- There are various tools (for pretty much all languages) to tell you how long your code is taking.
- They often track which functions are called, how many times, and how long each one takes.
- profile/cProfile can help with this.

Performance

- Python is an interpreted language. This means things in loops are very slow!
- Look at simple_hist_2d.py. This is a possibly relevant case where you take a list of points and grid them into a 2D array.
- How does this speed compare with anything you might expect?

C

- Compiled code is usually much faster, at least where loops are involved.

- We can link C to python, so python can call C libraries.

- ctypes is one way to do this. There are others, but ctypes widely present and requires no setup.

- Look at simple_hist_2d_from_c.py and hist2d_c.c as an example. We win by a factor of 100.