Lecture 8 - High Level Language Optimizations

DSE 512

Drew Schmidt
2022-02-17

From Last Time

- Homework is out
- Haven't made the slack channel yet
- Questions?

Basics

Profiling

- Real profiling comes later
- For now, we only need simple timing
- "Wall clock" timers
- Other kinds of timers exist…

Wallclock Timing

Python

```python
import time

t0 = time.perf_counter()
time.sleep(1.2)
t1 = time.perf_counter()
t1 - t0
```

```
## 1.2235700309975073
```

R

```r
system.time(Sys.sleep(1.2))[3]
```

```
## elapsed
##   1.202
```

Python Timer Class

```python
import time

class Timer(object):
  def start(self):
    self.t0 = time.perf_counter()

  def stop(self):
    t1 = time.perf_counter()
    print(t1-self.t0)
```

```python
t = Timer()

t.start()
time.sleep(.63)
t.stop()
```

```
## 0.6443845240864903
```

Major HLL Strategies

- Compilers and Flags
- Fundamental types
- Use efficient kernels/packages
- Vectorization
- HLL Compilers: JIT and bytecode

Compilers and Flags

- You can compile Python/R with different compilers and flags
  - `icc` instead of `gcc`
  - `-O3` instead of `-O2`
  - AND MANY MORE
- Pros:
  - Essentially **free** performance (if you can figure it out)
  - No additional effort!
- Cons:
  - Not always easy to do
  - Aggressive flags can break things
- We won't deal with this further

Fundamental Types

- Floating point: `float` (32-bit) and `double` (64-bit)
- Most computation (R, Python) automatically in double precision
- 32-bit Float:
  - half the memory
  - twice as fast (roughly)
  - not as accurate
- Python
  - Well supported in numpy
  - `np.random.rand(3).astype('f')`
  - `np.array([1, 2, 3, 4], dtype='f')`
- R
  - float package https://cran.r-project.org/package=float
  - fmlr package https://hpcran.org/packages/fmlr/index.html

Efficient Kernels

- Use existing functions/methods/packages that solve problems *well*.
  - Don't write your own linear model fitter
  - Don't write your own matrix multiplication
  - Don't write your own deep learning framework
- Exceptions
  - Interface work
  - Learning
  - Having a *genuinely* new/different approach

Vectorization

- A specific kind of use of efficient kernels
- Not rigorously defined (sometimes means something different!)
- Simplest explanation: avoiding loops
- Trades memory for runtime performance
- Operates on a vector of inputs
  - Can be a transformation: vector(s) in → vector out (e.g. `x+1`)
  - Can be a reduction: vector(s) in → number out (e.g. sum operator, dot product, etc)
  - Not strictly limited to these patterns…

Vectorization (Pseudocode)

Non-Vectorized

```
s = initialize()
for (i in 1:n)
  s += my_operation()
```

How many numbers do we store?

Vectorized

```
x = my_operation(n)
sum(x)
```

And here?

Vectorization Example: Dot Product

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Non-Vectorized

```
d = 0
for i in range(len(a)):
  d += a[i]*b[i]
```

How many numbers do we store?

Vectorized

```
numpy.dot(a, b)
```

And here?

Bytecode

- Your computer accepts a specific kind of instructions
- Those look nothing like R/Python instructions
- Bytecode is like machine code for interpreters
- C $\xrightarrow{\text{compiler}}$ machine code
- R and Python are C programs
- So R/Python $\xrightarrow{\text{compiler}}$ C $\xrightarrow{\text{compiler}}$ machine code?

Machine Code

```c
#include <stdio.h>

int main()
{
  printf("hi\n");
  return 0;
}
```

```
gcc -g hello.c -o hello
```

```
(gdb) disass main
Dump of assembler code for function main:
  0x0000000000001149 <+0>:     endbr64
  0x000000000000114d <+4>:     push   %rbp
  0x000000000000114e <+5>:     mov    %rsp,%rbp
  0x0000000000001151 <+8>:     lea    0xeac(%rip),%rd
  0x0000000000001158 <+15>:     callq  0x1050 <puts@p
  0x000000000000115d <+20>:     mov    $0x0,%eax
  0x0000000000001162 <+25>:     pop    %rbp
  0x0000000000001163 <+26>:     retq
End of assembler dump.
```

Machine Code

```c
#include <stdio.h>

int main()
{
  printf("hi\n");
  return 0;
}
```

```
gcc -g -O2 hello.c -o hello
```

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000000000001060 <+0>:     endbr64
   0x0000000000001064 <+4>:     sub    $0x8,%rsp
   0x0000000000001068 <+8>:     lea    0xf95(%rip),%rd
   0x000000000000106f <+15>:    callq  0x1050 <puts@p
   0x0000000000001074 <+20>:    xor    %eax,%eax
   0x0000000000001076 <+22>:    add    $0x8,%rsp
   0x000000000000107a <+26>:    retq
End of assembler dump.
```

Python Bytecode

```python
def hello():
  print('hi')

import dis
dis.dis(hello)
```

```
##   2           0 LOAD_GLOBAL              0 (print)
##               2 LOAD_CONST               1 ('hi')
##               4 CALL_FUNCTION            1
##               6 POP_TOP
##               8 LOAD_CONST               0 (None)
##              10 RETURN_VALUE
```

R Bytecode

```r
hello = function() print("hi")
hello = compiler::cmpfun(hello)
compiler::disassemble(hello)
```

```
## list(.Code, list(12L, GETFUN.OP, 1L, PUSHCONSTARG.OP, 3L, CALL.OP,
##     0L, RETURN.OP), list(print("hi"), print, structure(c(1L,
## 9L, 1L, 30L, 9L, 30L, 1L, 1L), srcfile = <environment>, class = "srcref"),
##     "hi", structure(c(NA, 0L, 0L, 0L, 0L, 0L, 0L, 0L), class = "expressionsIndex"),
##     structure(c(NA, 2L, 2L, 2L, 2L, 2L, 2L, 2L), class = "srcrefsIndex")))
```

JIT

- Just-In-Time
- Compilation occurs during execution
- Both modern Python and R use JIT for bytecode
- Other kinds of JITs exist --- more later

Calculating Means in Python

Mean Calculation

- *Many* ways to do this
  - Stable/unstable
  - Online
  - Mean + variance
  - Online mean + variance
- Some have numerical problems that go *well* beyond the scope of this course
- Good place to start https://en.wikipedia.org/wiki/Kahan_summation_algorithm
- Real example usage https://github.com/wrathematics/proginfo

Generate Some Random Data

```python
import random
random.seed(1234)
import numpy as np
n = 100000
x = np.random.rand(n)
```

Naive Mean

```python
def mean(x):
  s = 0.0
  n = len(x)
  for i in range(n):
    s += x[i]
  return s / n
```

```python
t.start()
mean_x = mean(x)
t.stop()
```

```
## 0.04085645999292937744
```

Vectorized Mean

```
def mean_numpy(x):
  return np.sum(x) / len(x)
```

```
t.start()
mean_x = mean_numpy(x)
t.stop()
```

```
## 0.009058568044565618
```

NUMBA

- A different kind of JIT compiler
- One of the cooler Python features!
- Converts some Python into machine code using LLVM
  - some core operations
  - NumPy
  - potentially good for loops of scalar/vector/matrix ops
- Somewhat similar to armacmp https://github.com/dirkschumacher/armacmp

NUMBA Mean

```python
from numba import jit

@jit(nopython = True)
def mean_numba(x):
  s = 0.0
  n = len(x)
  for i in range(n):
    s += x[i]
  return s / n
```

```python
t.start()
mean_x = mean_numba(x)
t.stop()
```

```
## 0.2743876969907433
```

```python
t.start()
mean_x = mean_numba(x)
t.stop()
```

```
## 0.007991217076778412
```

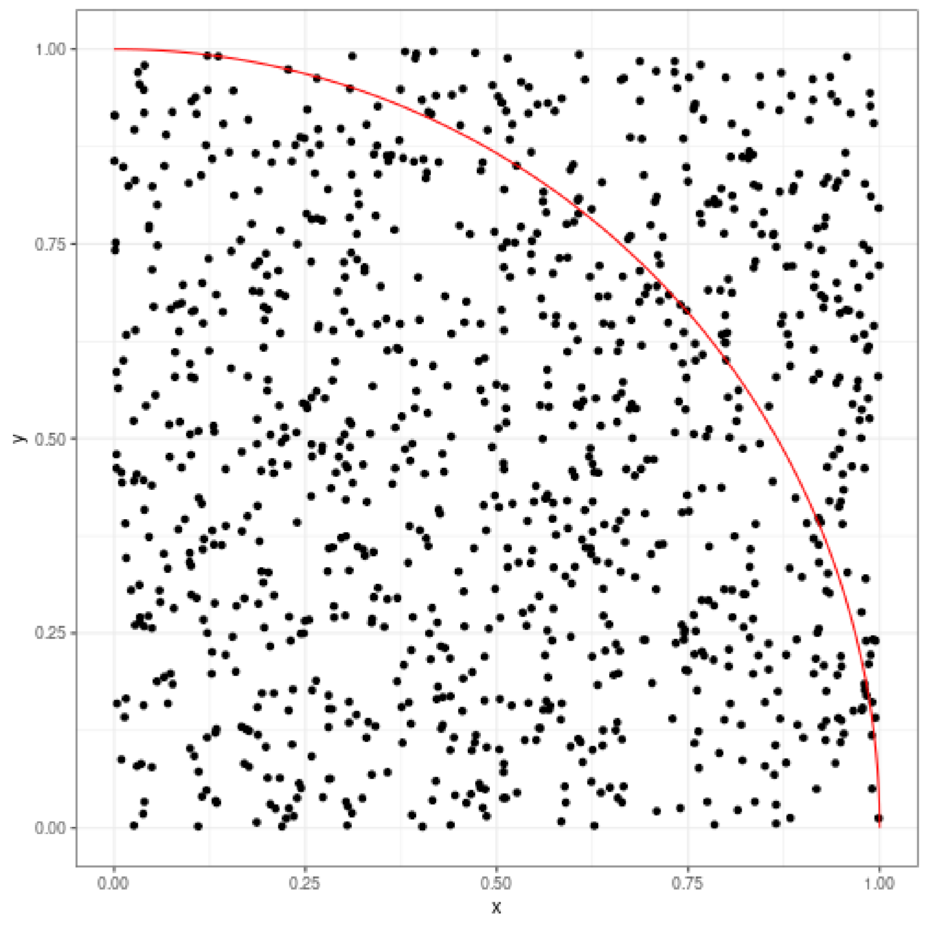
Calculating π

Monte Carlo π Simulation

$\text{Area} = \pi r^2$

```
set.seed(1234)
n = 1000
x = runif(n)
y = runif(n)

circ_pts = seq(0, pi/2,length.out=100)
library(ggplot2)
g = ggplot(data.frame(x=x, y=y), aes(x, y)
  theme_bw() +
  geom_point() +
  annotate("path",
   x = 0 + 1*cos(circ_pts),
   y = 0 + 1*sin(circ_pts),
   color="red"
  )
```



Estimating π: Naive Implementation

```r
pi_sim = function(n=1e6, seed=1234)
{
  set.seed(seed)
  x = runif(n)
  y = runif(n)

  s = 0
  for (i in 1:n)
  {
    if (x[i]*x[i] + y[i]*y[i] < 1)
      s = s + 1
  }

  4 * s / n
}
```

```r
system.time({
  pi_est <- pi_sim()
})[3]
```

```
## elapsed
##   0.236
```

```r
pi_est
```

```
## [1] 3.143128
```

```r
pi_sim_cmp <- compiler::cmpfun(pi_sim)
system.time({
  pi_est <- pi_sim_cmp()
})[3]
```

```
## elapsed
##   0.238
```

Estimating π: Vectorized

```r
pi_sim_vec = function(n=1e6, seed=1234)
{
  set.seed(seed)
  x = runif(n)
  y = runif(n)
  4 * sum(x*x + y*y < 1) / n
}
```

```r
system.time({
  pi_est <- pi_sim_vec()
})[3]
```

```
## elapsed
##   0.045
```

```r
pi_est
```

```
## [1] 3.143128
```

Estimating π: Python

- Discussed last year $\rightarrow$
- Ungraded Homework:
  - Create naive impl
  - Create numpy impl
  - Use numba impl
  - Time them all
  - Experiment with different sized samples

```python
import random
from numba import jit

@jit(nopython = True)
def monte_carlo_pi(nsamples):
  acc = 0
  for i in range(nsamples):
    x = random.random()
    y = random.random()
    if (x**2 + y**2) < 1:
      acc += 1
  return 4 * acc / nsamples
```

Questions?