CoE 163

Computing Architectures and Algorithms

03a: High-Level Optimization

MAXIMIZING ALGORITHMS

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

- Donald Knuth, "The Art of Computer Programming"

MAXIMIZING ALGORITHMS

It is much more important to create *correct* code than efficient code. Write correctly now, speed it up later.

Optimizations, however, are useful especially if a lot of the running time is spent on some piece of code.

AMDAHL'S LAW

- Expression for the maximum expected improvement of the whole system if a part of it is optimized
- Usually used in parallel programming, but we can still use it for our "non-parallel" programs

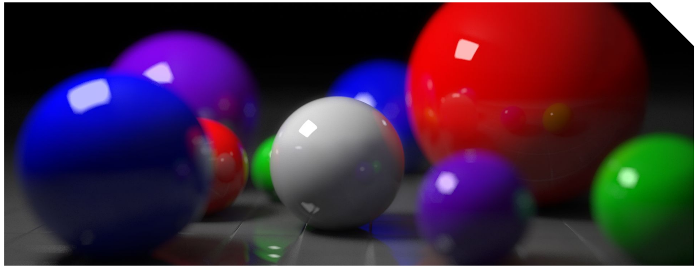
AMDAHL'S LAW

$$S = \frac{1}{(1 - f_E) + \frac{f_E}{f_I}}$$

If f_E (percent) of the code has been sped-up by f_I (times), then the whole program will have a maximum speedup of S (times).

AMDAHL'S LAW

Suppose that we have a raytracer program with the intersection algorithm (around 40% of the whole program) sped-up by 5 times.



AMDAHL'S LAW

$$
\begin{aligned}
S &= \frac{1}{(1 - f_E) + \frac{f_E}{f_I}} \\
&= \frac{1}{(1 - 0.4) + \frac{0.4}{5}} \\
&\approx 1.47
\end{aligned}
$$

Was it worth it? Yes, if that code is commonly used.

GAINING SPEED-UPS

Knowing some basic code optimizations will come a long way in squeezing out less time from your code.

Code for correctness, but make obvious optimizations when opportunity comes.

C/C++ SPEED HACKS

- Use bit shift when multiplying or dividing by two
- Simplify math expressions to reduce the number of operations
- Take advantage of short-circuit logic because conditionals are expensive

C/C++ SPEED HACKS

- Prefer pre-increment over post-increment
- Prefer iteration over recursion since function calls use the stack pointer
- Prefer pass-by-reference over pass-by-value
- … a lot more!

PYTHON SPEED HACKS

- Some code parts may benefit from being coded into C and linked into Python
- Convert loops to list comprehensions or generators
- Take advantage of short-circuit logic
- … and many more!

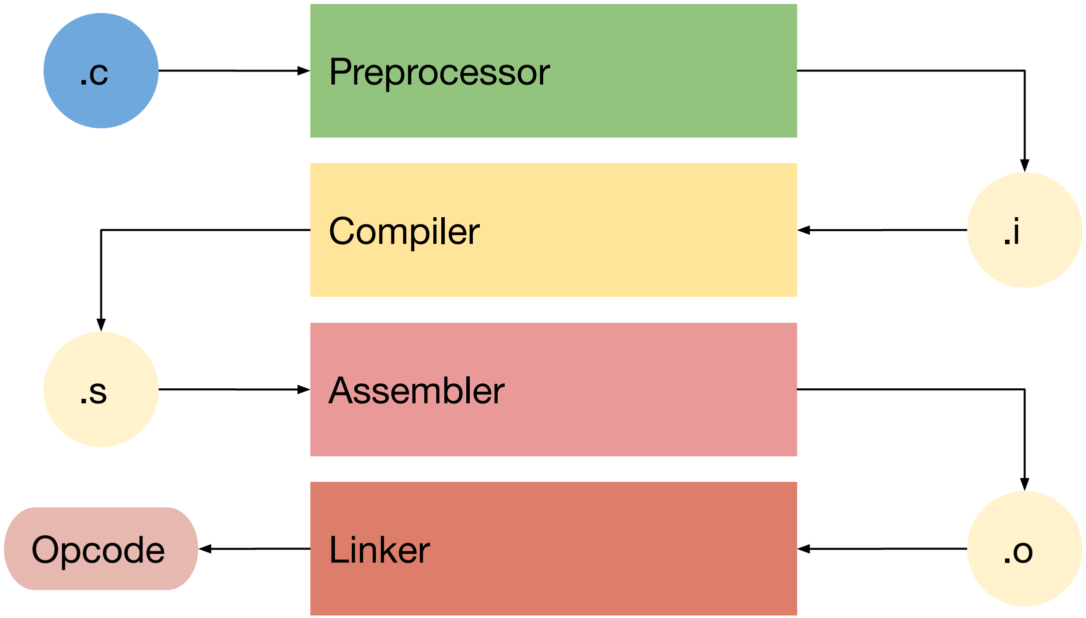
HOW WERE THEY DISCOVERED?

- Knowledge of assembly and the compiler
- Knowledge of computer architecture and microarchitectures
- Time and space profiling

GCC

- The GNU Compiler Collection (GCC) is an optimizing compiler
- GCC has initially supported only C in 1987, but can now compile Go and D, among others
- It is an essential part of the GNU toolchain

GCC COMPILATION PIPELINE

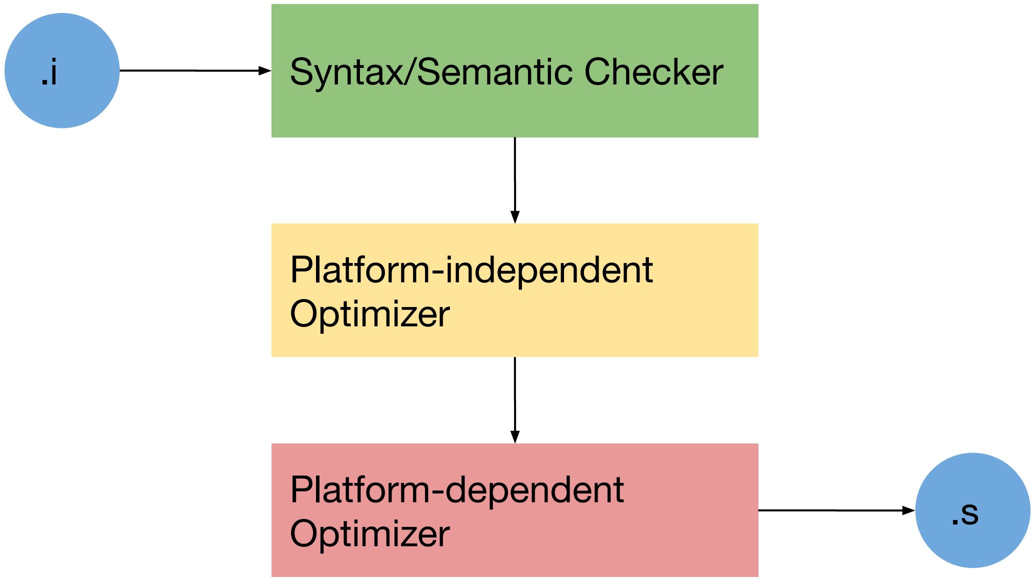


GCC COMPILER

GCC compiles code in three-stages

- Front - syntax checking and parsing to an intermediate representation
- Middle - platform-independent optimization
- Back - platform-dependent optimization and conversion to assembly

GCC COMPILER



GCC: ASSEMBLY

With the structure of GCC, it is possible to generate the intermediate preprocessor and assembly codes.

We can investigate how our code works at the low-level by reading the resulting assembly code.

GCC: ASSEMBLY

```c
int main() {
    int x[] = {1, 6, 3};
    return x[0] + x[1] + x[2];
}
```

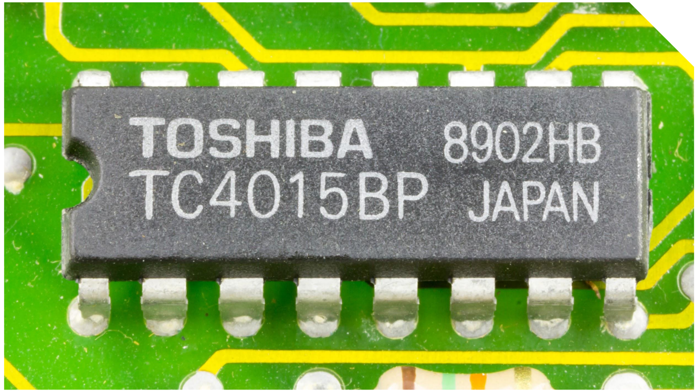
```
gcc -S arrays.c
```

Intel ASM

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-12], 1
mov     DWORD PTR [rbp-8], 6
mov     DWORD PTR [rbp-4], 3
mov     edx, DWORD PTR [rbp-12]
mov     eax, DWORD PTR [rbp-8]
add     edx, eax
mov     eax, DWORD PTR [rbp-4]
add     eax, edx
pop     rbp
ret
```

CONSIDER...

Let's verify whether bit shifting is faster than division with a power of 2.



INTEGER DIVISION

```
int x = 16;
int y = x / 4;
```

```
gcc -S div2.c
```

Intel ASM

```
mov  DWORD PTR [rbp-4], 16
mov  eax, DWORD PTR [rbp-4]
lea  edx, [rax+3]
test     eax, eax
cmovs    eax, edx
sar  eax, 2
mov  DWORD PTR [rbp-8], eax
```

Extra test for integer division

BIT SHIFTING BY TWO

```
int x = 16;
int y = x >> 2;
```

```
gcc -S div2.c
```

Intel ASM

```
mov  DWORD PTR [rbp-4], 16
mov  eax, DWORD PTR [rbp-4]
sar  eax, 2
mov  DWORD PTR [rbp-8], eax
```

No extra test!

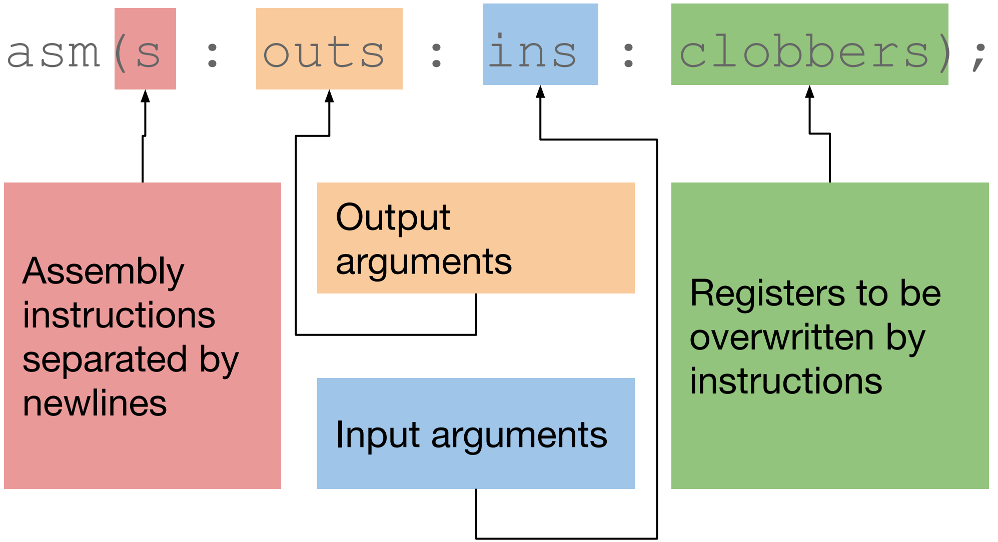
DIVISION BY POWER OF TWO

- When using integer division, the code still has to check whether the number is less than 0
- Load instructions are slow, but bitwise operations are fast
- Bit shifting is marginally faster than division by a power of two

ASSEMBLY IN C/C++

- GCC supports writing and compiling of assembly code within C/C++
- This is useful for systems development where some sections would run faster in assembly

INLINE ASSEMBLY IN C/C++



ASSEMBLY IN C/C++

```
int a = 3, b = 3, c;

asm(
    "mov %1, %%eax\n"
    "mov %2, %%ebx\n"
    "add %%eax, %%ebx\n"
    "mov %%ebx, %0\n"
    : "=r" (c)
    : "r" (a), "r" (b)
    : "%eax", "%ebx"
 );
```

Add a and b

AT&T ASM

ASSEMBLY IN C/C++

Although the feature is powerful, it is relatively easier to write a whole function in assembly.

It is also possible to write the assembly code separately as an `.s` file.

ASSEMBLY IN C/C++

```
long add_me(long in, long in2); /* Prototype */

asm( /* Assembly function body */
    "add_me:\n"
    "    mov %rdi, %rax\n"
    "    add %rsi, %rax\n"
    "    ret\n"
);

int main(void) {
   return add_me(3, 5);
}
```

Add in and in2

AT&T ASM

TIME PROFILING IN C/C++

- Several binaries and software can be used to profile C/C++ Code
- Simplest is to use the built-in profiler that came with GCC

TIME PROFILING IN C/C++

gprof

- Old profiler that uses statistical sampling to measure runtime
- Generates a decent report on the runtime per function of a program
- May be inaccurate since sampling time is usually 0.01s

TIME PROFILING IN C/C++

gprof

- Compile code as normal with flags `-pg`
- Run programas normal and it will generate profile data named `gmon.out` in the directory where you are running the program
- Run gprof with the executable and profile data as arguments
- gprof generates a report on standard output - use redirection to output into a file

```
$ gcc add_me.c -o add_me -pg
$ ./add_me
$ gprof add_me gmon.out
```

TIME PROFILING IN C/C++

gprof - flat profile*

- Shows time spent running each function of a program
- Broken down into cumulative, number of calls, and percentage of program runtime executing such function

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
[...]
```

* not the profile of the add_me function

TIME PROFILING IN C/C++

gprof - call graph*

- Shows time spent running a function and functions that it called during execution
- Useful for knowing a bit more information on where a program spends most of its runtime

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05
seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
[...]
```

* not the profile of the add_me function

TIME PROFILING IN C/C++

perf

- Newer general-purpose profiler for the Linux kernel
- Has a command line interface for viewing reports and even assembly code

TIME PROFILING IN C/C++

perf

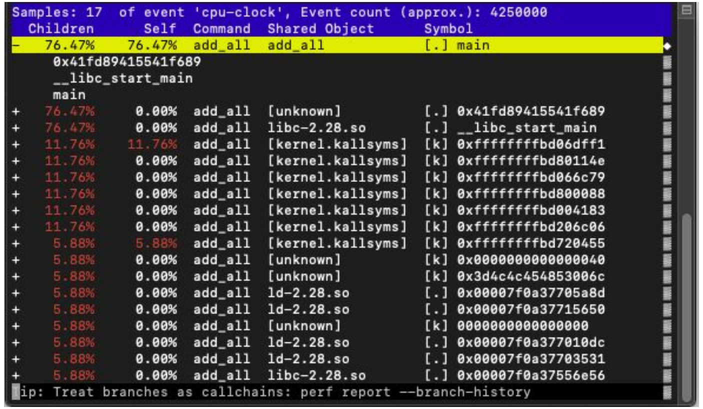
- Compile code as normal
- Run program through perf, and it will generate profile data named `perf.data` in the directory where you are running the program
- Run `perf report` to view the report

```
$ gcc add_me.c -o add_me
$ perf record -g ./add_me
$ perf report
```

TIME PROFILING IN C/C++

perf - call graph

- Shows percentage time spent running a function and functions that it called during execution
- Shows the different libraries and functions called during execution

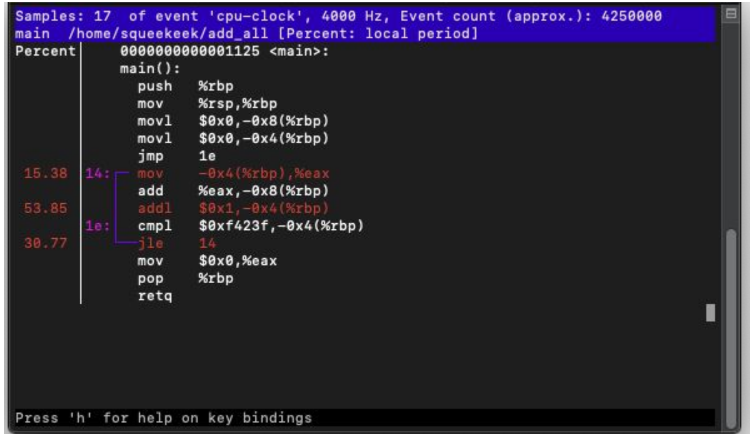


```
Samples: 17  of event 'cpu-clock', Event count (approx.): 4250000
  Children      Self  Command  Shared Object        Symbol
-   76.47%    76.47%  add_all  add_all              [.] main
     0x41fd89415541f689
     __libc_start_main
     main
+   76.47%     0.00%  add_all  [unknown]            [.] 0x41fd89415541f689
+   76.47%     0.00%  add_all  libc-2.28.so         [.] __libc_start_main
+   11.76%    11.76%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd06dff1
+   11.76%     0.00%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd80114e
+   11.76%     0.00%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd066c79
+   11.76%     0.00%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd800088
+   11.76%     0.00%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd004183
+   11.76%     0.00%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd206c06
+    5.88%     5.88%  add_all  [kernel.kallsyms]    [k] 0xffffffffbd720455
+    5.88%     0.00%  add_all  [unknown]            [k] 0x0000000000000040
+    5.88%     0.00%  add_all  [unknown]            [k] 0x3d4c4c454853006c
+    5.88%     0.00%  add_all  ld-2.28.so           [.] 0x00007f0a37705a8d
+    5.88%     0.00%  add_all  ld-2.28.so           [.] 0x00007f0a37715650
+    5.88%     0.00%  add_all  [unknown]            [k] 0000000000000000
+    5.88%     0.00%  add_all  ld-2.28.so           [.] 0x00007f0a377010dc
+    5.88%     0.00%  add_all  ld-2.28.so           [.] 0x00007f0a37703531
+    5.88%     0.00%  add_all  libc-2.28.so         [.] 0x00007f0a37556e56
Tip: Treat branches as callchains: perf report --branch-history
```

TIME PROFILING IN C/C++

perf - disassembler

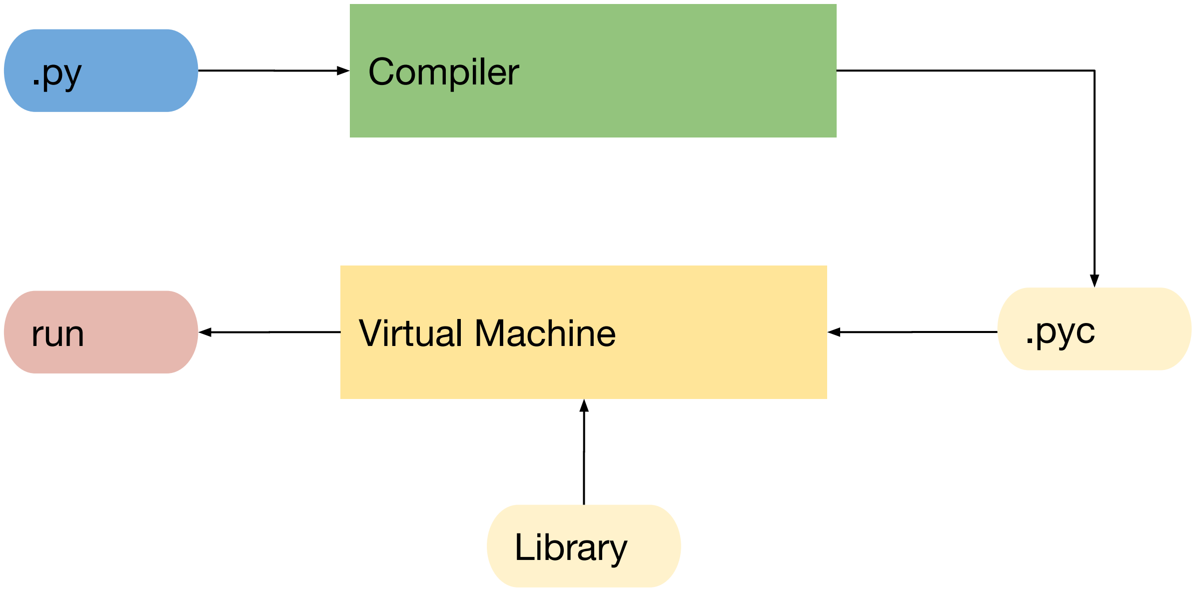
- Shows percentage time spent running each assembly instruction during execution
- Useful for knowing a bit more information on where a program spends most of its runtime



CPYTHON

- CPython is the reference implementation of Python since 1994
- It is an interpreter with an internal helper compiler
- It can either execute programs on-the-fly, or compile it into a platform-independent bytecode

CPYTHON INTERPRETATION PIPELINE



CPYTHON: "ASSEMBLY"

Since Python is interpreted, it generates a platform-independent bytecode instead of assembly code.

We can investigate how our code works at the intermediate level by reading the resulting bytecode.

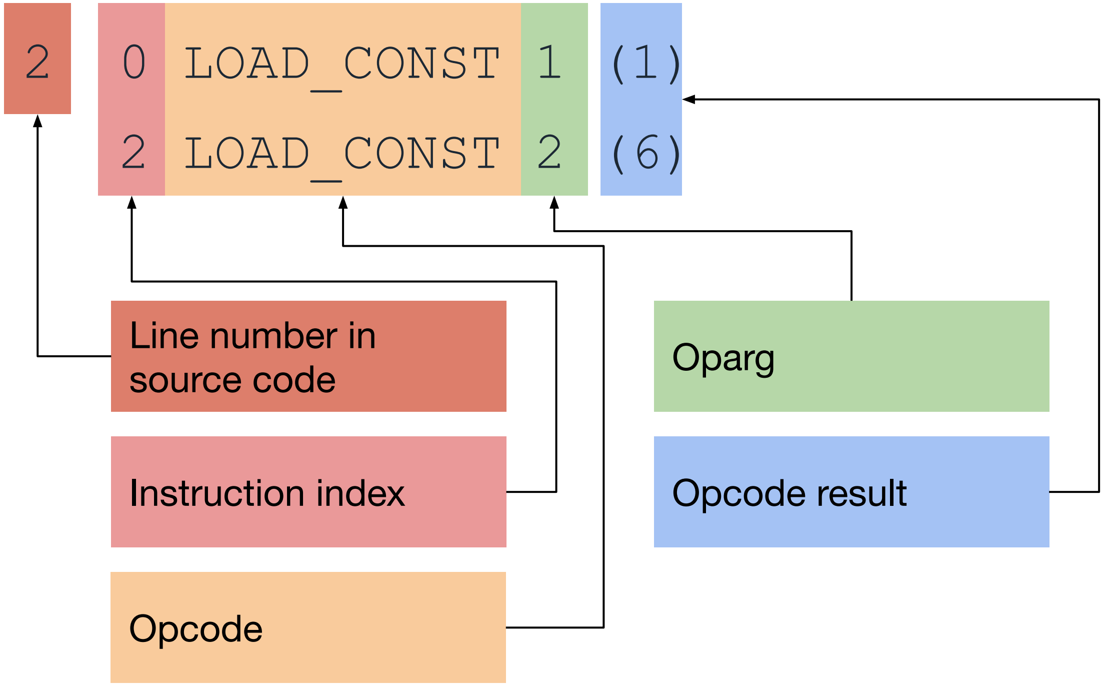
CPYTHON: BYTECODE

```
def main():
    x = [1, 6, 3]
    return x[0] + x[1] + x[2]
```

```
dis.dis(main)
```

```
2           0 LOAD_CONST               1 (1)
            2 LOAD_CONST               2 (6)
            4 LOAD_CONST               3 (3)
            6 BUILD_LIST               3
            8 STORE_FAST               0 (x)
3          10 LOAD_FAST                0 (x)
           12 LOAD_CONST               4 (0)
           14 BINARY_SUBSCR
           [...]
           32 RETURN_VALUE
```

CPYTHON: BYTECODE



CONSIDER...

Let's check the fastest way to add all the numbers in a list.

NORMAL LOOP

```
sum_all = 0

for i in range(len(num_list)):
    sum_all += num_list[i]
    return sum_all
```

```
dis.dis(add_all)
```

```
2           0 LOAD_CONST               1 (0)
            2 STORE_FAST               1 (sum_all)
4           4 SETUP_LOOP              34 (to 40)
              [...]
           20 STORE_FAST               2 (i)
5          22 LOAD_FAST                1 (sum_all)
              [...]
           32 STORE_FAST               1 (sum_all)
7          34 LOAD_FAST                1 (sum_all)
              [...]
           42 RETURN_VALUE
```

Too many operations!

FUNCTOOLS REDUCE

```
return reduce(lambda a, b: a + b, num_list)
```

```
dis.dis(add_all)
```

```
2           0 LOAD_GLOBAL              0 (reduce)
            2 LOAD_CONST               1 (...)
            4 LOAD_CONST               2 (...)
            6 MAKE_FUNCTION            0
            8 LOAD_FAST                0 (num_list)
           10 CALL_FUNCTION            2
           12 RETURN_VALUE
```

Much better!

BUILT-IN SUM

```
return sum(num_list)
```

```
dis.dis(add_all)
```

```
2           0 LOAD_GLOBAL              0 (sum)
            2 LOAD_FAST                0 (num_list)
            4 CALL_FUNCTION            1
            6 RETURN_VALUE
```

This is the best.

SUMMING A LIST

- Constructing a for-loop requires set-up on the interpreter, and is relatively slow
- “Idiomatic” code, such as list comprehensions, runs faster than for-loops
- Built-in functions are the fastest due to them being globally accessible and leveraging a C backend (for CPython)

TIME PROFILING IN PYTHON

- There are several built-in modules and functions in Python for the purpose
- Simplest is to record the start and end time of executing a code section

TIME PROFILING IN PYTHON

timeit module

- Simplest timer for very small code snippets
- Runs the snippet 1 million times by default
- All code should be fed as strings

```
from timeit import timeit

timeit(‘’’
sum(int_list)
‘’’, setup=’’’
import numpy as np; int_list = np.random.randint(1, 100,
(1000,));
‘’’)
```

TIME PROFILING IN PYTHON

time module

- Import the time library and get the time at appropriate instances of the program
- Simple and fast to use

```
import time
import numpy as np

start_t = time.time()

int_list = np.random.randint(1, 100, (1000,))
add_all(int_list)

end_t = time.time()
print(f'Time elapsed: {end_t - start_t}s')
```

TIME PROFILING IN PYTHON

cProfile module

- Deterministic profiler with advanced break-down of time elapsed for each component
- Has an accuracy only up to 0.001s

```
import cProfile
import numpy as np

int_list = np.random.randint(1, 100, (1000,))
cProfile.run(‘’’
for i in range(1000000):
    add_all(int_list)
‘’’)
```

TIME PROFILING IN PYTHON

cProfile output

- Broken down by (sub)functions called
- Contains runtime in seconds and number of calls to that function during the whole profiling

```
      2000003 function calls in 1.335 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1000000    0.938    0.000    1.026    0.000 <ipython-input-38-94ea3e02d399>:1(add_all)
      1    0.309    0.309    1.335    1.335 <string>:2(<module>)
      1    0.000    0.000    1.335    1.335 {built-in method builtins.exec}
1000000    0.089    0.000    0.089    0.000 {built-in method builtins.len}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

TIPS

- Make the common case fast
- Program in assembly as a last resort
- Premature optimization is bad, but obvious optimization should be done
- Optimization takes twice time as normal programming

TIPS

- Profile different implementations to determine the fastest one
- Find the best profiler, or profiling strategy, according to your needs

RESOURCES

- ~~Raytracer~~ C/C++ optimization tips from Clemson University
- C/C++ to Assembly optimization resources from Agner Fog
- Compiler Explorer to check compilation results in C, C++, Python, and many more

RESOURCES

- Gprof resource from the University of Utah
- Short blog on perftools usage

CoE 163

Computing Architectures and Algorithms

03a: High-Level Optimization