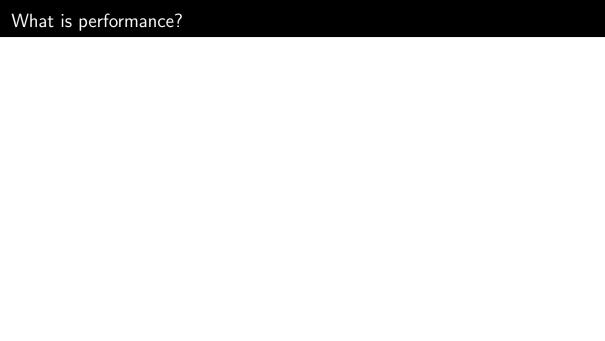
Performance and Languages

Troels Henriksen

22nd of September, 2021



What is performance?

- How **fast** you solve a problem.
- How **few resources** you use to solve a problem.

What is performance?

- How **fast** you solve a problem.
- How **few resources** you use to solve a problem.

Important: the program must still be correct—it's easy to make a very fast program if it doesn't have to produce the right result.

What is performance?

- How **fast** you solve a problem.
- How **few resources** you use to solve a problem.

Important: the program must still be correct—it's easy to make a very fast program if it doesn't have to produce the right result.

There's more to performance than asymptotic complexity!

Constant factors matter

- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
 - ▶ algorithm, data representations, procedures, and loops
- Must understand system to optimise performance:
 - ► How programs are compiled and executed
 - ► How modern processors + memory systems operate
 - ► How to measure program performance and identify bottlenecks
 - ► How to improve performance without destroying code modularity and generality

Constant factors matter

- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
 - ▶ algorithm, data representations, procedures, and loops
- Must understand system to optimise performance:
 - ► How programs are compiled and executed
 - ► How modern processors + memory systems operate
 - ► How to measure program performance and identify bottlenecks
 - ▶ How to improve performance without destroying code modularity and generality

Let's look at why C is often considered a "fast language", and what that means.

Roughly two kinds of languages

Compiled languages are transformed to machine code before execution (e.g. C) Interpreted languages are run directly by a software *interpreter* (e.g. Python)

Roughly two kinds of languages

Compiled languages are transformed to machine code before execution (e.g. C) Interpreted languages are run directly by a software *interpreter* (e.g. Python)

Pedantic disclaimer

Compilation/interpretation is strictly a property of *implementations*, not *languages*.

- You could have a C interpreter or Python compiler
- But most (not all!) languages are built with a specific implementation technique in mind
- A few languages (Lisp, JavaScript) have lots of *very* different implementations

Roughly two kinds of languages

Compiled languages are transformed to machine code before execution (e.g. C) Interpreted languages are run directly by a software *interpreter* (e.g. Python)

Pedantic disclaimer

Compilation/interpretation is strictly a property of *implementations*, not *languages*.

- You could have a C interpreter or Python compiler
- But most (not all!) languages are built with a specific implementation technique in mind
- A few languages (Lisp, JavaScript) have lots of *very* different implementations

We teach you the big picture—the details are always more complicated in practice!

Tradeoffs

- Compiled languages
 - + Almost always faster
 - Require compilation after every change
 - Usually cannot run program fragments in isolation
 - Tend to have more restrictions (e.g. static typing)
 - Much more difficult to implement

Tradeoffs

Compiled languages

- + Almost always faster
- Require compilation after every change
- Usually cannot run program fragments in isolation
- Tend to have more restrictions (e.g. static typing)
- Much more difficult to implement

Interpreted languages

- Usually slow
- + Can run immediately
- + Can easily run fragments (e.g. single functions) in isolation
- + Much easier to implement

Tradeoffs

- Compiled languages
 - + Almost always faster
 - Require compilation after every change
 - Usually cannot run program fragments in isolation
 - Tend to have more restrictions (e.g. static typing)
 - Much more difficult to implement
- Interpreted languages
 - Usually slow
 - + Can run immediately
 - + Can easily run fragments (e.g. single functions) in isolation
 - + Much easier to implement

Let us look at the scale of the overhead.

The Collatz conjecture

$$f(n) = \left\{ \begin{array}{ll} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{array} \right\}$$

- **Conjecture:** if we apply this function to some number greater than 1, we will eventually reach 1
- To disprove this conjecture, we only need a single counter-example that goes into a cycle instead
- People write programs to investigate the behaviour of this sequence

Listing 1: collatz.pv

```
import sys
def collatz(n):
    i = 0
    while n != 1:
        if n % 2 == 0.
            n = n // 2
        else:
            n = 3 * n + 1
        i = i + 1
    return i
k = int(sys.argv[1])
for n in range(1, k):
    print(n, collatz(n))
```

```
Listing 2: collatz.c
```

```
#include <stdio.h>
#include < stdlib . h>
#include <assert.h>
int collatz(int n) {
  int i = 0:
  while (n != 1) {
    if (\hat{n} \% 2 = \hat{0}) \{ n = n / 2; \}
    i++:
  return i:
int main(int argc, char** argv) {
  assert(argc == 2):
  int k = atoi(argv[1]):
  for (int n = 1; n < k; n++) {
    printf("%d_%d\n", n, collatz(n));
```

```
$ time python3 ./collatz.py 100000 >/dev/null
```

0m0.002s

sys

```
$ time python3 ./collatz.py 100000 >/dev/null
real
        0m1.368s
user
        0m1.361s
        0m0.007s
sys
$ gcc collatz.c -o collatz
$ time ./collatz 100000 >/dev/null
real
        0m0.032s
        0m0.030s
user
```

0m0.002s

sys

```
$ time python3 ./collatz.py 100000 >/dev/null
real
        0m1.368s
       0m1.361s
user
        0m0.007s
sys
$ gcc collatz.c -o collatz
$ time ./collatz 100000 >/dev/null
real
        0m0.032s
        0m0.030s
user
```

Speedup:
$$\frac{1.308}{0.032} = 42.7$$

Why are interpreted languages slow?

To understand this, we must look at **how processors execute code**.

Register file
rax
rcx
rdx
rbx

- The CPU has a fixed set of **registers**
- Registers act as variables for machine code instructions
- Some registers are general purpose, others are special purpose
 - ► Program counter holds address of current instruction
 - Condition register holds results of comparisons
 - Stack pointer holds address of top of stack
- Exact registers and instructions depend on the **architecture**.
 - ► We teach you x86-64 (sometimes called AMD64)
 - ► Others are ARM, MIPS, SPARC, POWER...
 - ► On x86-64, most registers hold 64-bit quantities.

Compiling C to assembly

```
$ gcc mul.c -S
```

Listing 3: mul.c

```
long mul(long x, long y) {
  return x * y;
}
```

Listing 4: mul.s (extract)

```
mul:
.LFB0:
.cfi_startproc
movq %rdi, %rax
imulq %rsi, %rax
ret
.cfi_endproc
```

Calling convention:

- The two arguments stored in registers rdi and rsi.
- Return value stored in register rax.

A modern processor can execute billions of instructions per second.

Compiling Python to...?

The Python interpreter actually does compile Python to "opcodes".

This is "assembly code" for a virtual machine that is interpreted by a C program.

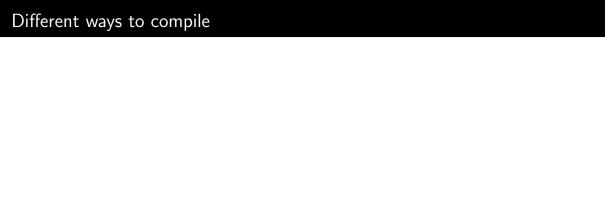
Compilation vs interpretation

All programs are interpreted when run!

- Machine code programs (e.g. generated by a C compiler) are run by an interpreter built in hardware.
- Python programs are run by an interpreter written in software, which must itself be interpreted by something else.
- Each level of interpretation adds overhead, but usually also *convenience*: productivity and correctness!

Combining interpretation and compilation

- Interpreted languages can be fast when
 - ► Most of the run-time is spent waiting data from files or network
 - ► They mostly call functions written in faster compiled languages
- Best of both worlds: flexibility of interpretation, and speed of C



Different ways to compile

To executable program collatz

- \$ gcc collatz.c -o collatz
 - Can be run directly

Different ways to compile

To executable program collatz

- \$ gcc collatz.c -o collatz
 - Can be run directly

To object file collatz.o

- \$ gcc collatz.c -c -o collatz.o
 - Can be linked with other object files
 - Can be processed further

Different ways to compile

To executable program collatz

- \$ gcc collatz.c -o collatz
 - Can be run directly

To object file collatz.o

- \$ gcc collatz.c -c -o collatz.o
 - Can be linked with other object files
 - Can be processed further

To shared object file libcollatz.so

- \$ gcc collatz.c -fPIC -shared -o libcollatz.so
 - Can be linked at run-time by a running program
 - How compiled programs support dynamic "plug-ins"

All output files contain fully compiled machine code.

Calling C from Python

Compiling C program to shared library

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

Listing 5: collatz-ffi.py

```
import ctypes
import sys

c_lib = ctypes.CDLL('./libcollatz.so')

k = int(sys.argv[1])
for n in range(1, k):
    print(n, c_lib.collatz(n))
```

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null
```

Speedup:
$$\frac{1.368}{0.165} = 8.2$$

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null
```

```
real 0m0.165s
user 0m0.163s
sys 0m0.003s
```

Speedup:
$$\frac{1.368}{0.165} = 8.2$$

- Slower than pure C by about 5×
- Faster if we made fewer "foreign" calls, but each took more time
- Ideal case is single foreign function call that operates on many values
- This is exactly how NumPy works!

NumPy performance

```
def f_python(v):
    for i in range(len(v)):
        v[i] = v[i]*2 + 3
```

def $f_numpy(v)$: return v * 2 + 3

Size of v	$\mathtt{f_python}$	${\tt f_numpy}$	Difference
1	0.01 ms	0.01 <i>ms</i>	$0.9 \times$
10	0.01 ms	0.01 ms	$1.4 \times$
100	0.1 ms	0.01 ms	$13.3 \times$
1000	0.98 <i>ms</i>	0.01 ms	$95.3 \times$
10000	9.96 <i>ms</i>	0.05 <i>ms</i>	$190.7 \times$
100000	98.59 <i>ms</i>	0.41 <i>ms</i>	$240.7 \times$

Conclusions

- Compiled languages tend to be fast, but less flexible
- Interpreted languages tend to be slower, but more flexible
- Best of both worlds: write computational primitives in fast languages, call them from slow languages
 - ► NumPy works like this