Profiling in *Python*

Markus Kunesch

faculty

Measure, don’t guess.

(the answer to every performance question on Stack Overflow)

Profile before optimising

(and during optimising, and after optimising…)

Simple profiling is easy

(you can integrate it into your workflow)

Contents

faculty

Questions

Profiling

Live measurement of a program's use of hardware resources:

- Walltime
- CPU time
- Memory

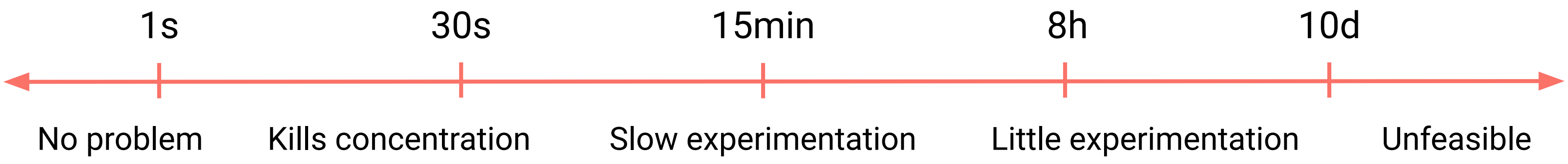
this talk

- I/O
- CPU cycles per instruction
- Cache hits/misses
- Branch mispredictions
- Utilisation of vector instructions
- ...

really cool, but less relevant for Python

Why profile?

- Helps you make your code faster
  - See why the code is slow
  - Potentially find simple performance gains
- Saves you time
  - Optimise only parts that matter
  - Stop optimising when there is little to be gained
  - Faster code will save you time



Why profile?

- Helps you make your code faster
    - See why the code is slow
    - Potentially find simple performance gains
- Saves you time
    - Optimise only parts that matter
    - Stop optimising when there is little to be gained
    - Faster code will save you time
- Helps you parallelise/thread the code
- Explains what the code does
- Helps you write good code

Demo: what is profiling?

(these slides, all code, and my bash history will be provided)

Profiling methods

Deterministic

Record the state of the execution whenever certain events occur.

High precision but potentially **high overhead**.

Statistical

Interrupt the execution repeatedly, record the state, get statistical results.

Tunable precision/overhead trade-off.

Time Sampling

Event-based (software)

Event-based (hardware)

Instrumentation

Profiling tools in Python

Deterministic

- profile
  Function timing, higher overhead
- cProfile
  C implementation of profile, same usage, much lower overhead
- yappi
  Function timing, works with threading
- line_profiler
  Line-by-line timing, implemented in C, potentially high overhead
- memory_profiler
  Memory usage, line-by-line
- etc.

Statistical

- pyflame
  Sampling profiler for Linux only, line-by-line and function timing, good for threading, low overhead.
- vmprof
  Sampling profiler, function or line-by-line timing, low-overhead
- python-flamegraph
  Sampling profiler, function or line-by-line timing, low-overhead
- plop
  Sampling profiler, function timing, low overhead
- etc.

Profiling tools in Python

Deterministic

- profile
  Function timing, higher overhead
- cProfile
  C implementation of profile, same usage, much lower overhead
- yappi
  Function timing, works with threading
- line_profiler
  Line-by-line timing, implemented in C, potentially high overhead
- memory_profiler
  Memory usage, line-by-line
- etc.

Statistical

- pyflame
  Sampling profiler for Linux only, line-by-line and function timing, good for threading, low overhead.
- vmprof
  Sampling profiler, function or line-by-line timing, low-overhead
- python-flamegraph
  Sampling profiler, function or line-by-line timing, low-overhead
- plop
  Sampling profiler, function timing, low overhead
- etc.

Profiling advice

- Profile a realistic setup
- Repeat after every optimisation step
- Keep it simple

- Profile on the target hardware
- To get accurate results:
  - Avoid heavily-loaded shared nodes
  - Repeat a few times
  - Choose low-overhead profiler for cheap functions/lines

Simple timing

Simple timing tools are sometimes enough:

time.time()

```
import time

start_time = time.time()
do_stuff()
print(
    time.time() - start_time, “s”
)

>> 10s
```

time (unix)

```
time python script.py

>> real    0m5.400s
   user    0m3.144s
   sys     0m0.121s
```

%timeit, %%time

IPython:

```
%timeit do_stuff()
>> 1 s ± 1.93 ms per loop
(mean ± std. dev. of 7 runs, 1
loop each)
```

```
%%time
do_stuff()
do_other_stuff()
>> CPU times: user 2.87 s,
sys: 16 ms, total: 2.88 s
Wall time: 4 s
```

Demo: work and sleep

Demo: Notes

- Example code demo_0/demo_0.py.
- Use time to demonstrate real, sys, user.
- Guess: which lines in work() are the most expensive? By how much?
- Run simple cProfile, visualise with gprof2dot & dot -Tpng
- Run line_profiler on the work function
- Do the same from within a Jupyter notebook using the magics.
- Profile memory usage using memory_profiler

Demo: video classifier

Demo: Notes

- Previously unseen code (demo_1)
- Let's not try to read it; let's profile it immediately
- Code takes too long to run. Options:
  - Profile cheaper setup (possible but risky)
  - Attach pyflame to running program
- See what the code does and use this to start reading
  - Terribly implemented function remove_edge() -> improve it
- Profile again, this time with cProfile. Visualise with gprof2dot & dot -Tpng

Part II:
Parallelisation and scaling

(using profiles to aid parallelisation, profiling parallel code)

Parallelisation is an *optimisation*

Profile before optimising

Profile before parallelising

Parallelisation and scaling

Parallelisation is an *optimisation*. Want to know:

- Is parallelisation worth it?
- How much faster will the code be?
- Where should the parallel region be?

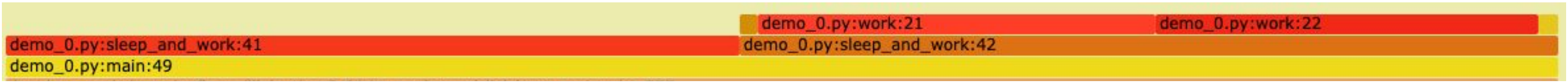
Measure, don’t guess.

Spotting threading opportunities

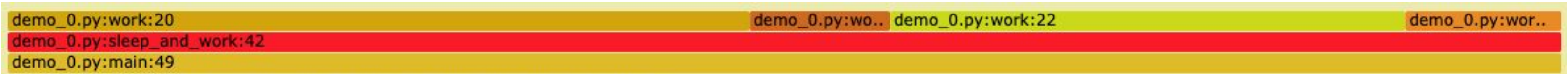
Threading: Expensive regions with low CPU time (GIL).

Run pyflame in real- and CPU-time mode.

Real time: (pyflame --threads)



CPU time, GIL only (pyflame -x)



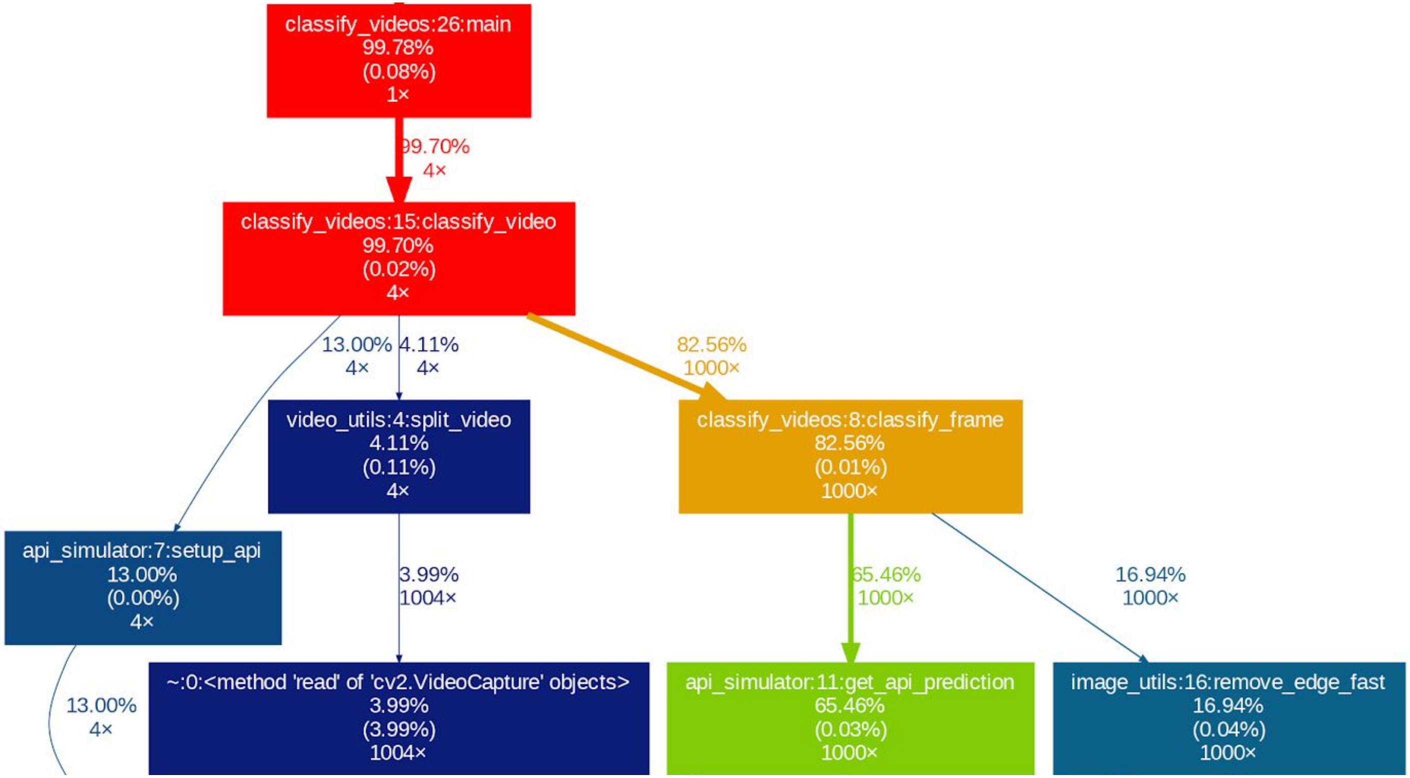
Spotting multiprocessing opportunities

Multiprocessing:
Expensive parallelisable regions (e.g. loops), high up in call graph.

Demo: video classifier

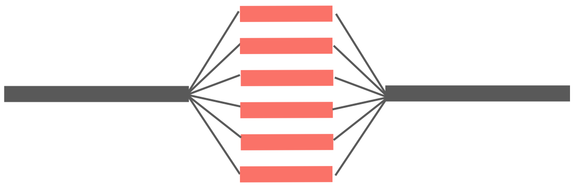
- Parallelise over frames or videos?

Demo: Notes

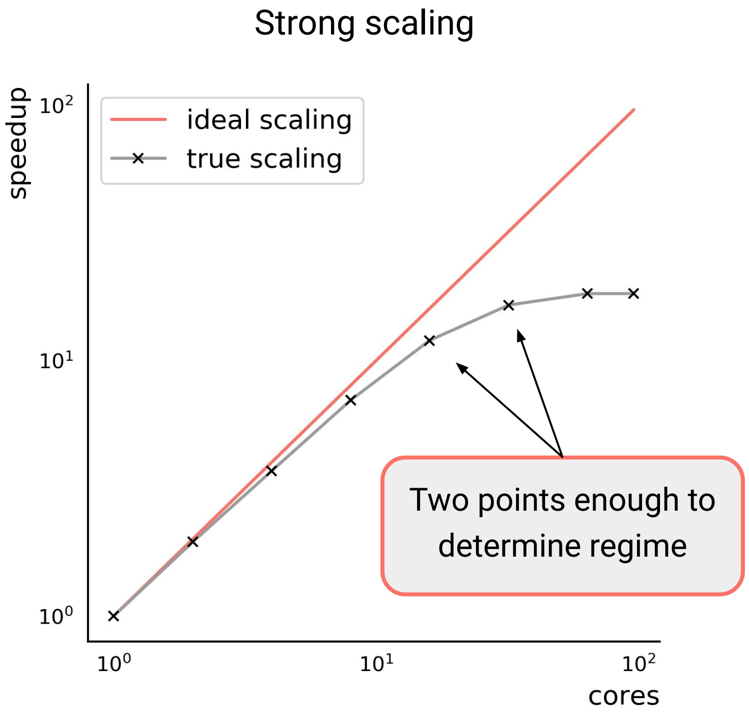

classify_videos:26:main
99.78%
(0.08%)
1×
99.70%
4×
classify_videos:15:classify_video
99.70%
(0.02%)
4×
13.00%
4×
4.11%
4×
82.56%
1000×
video_utils:4:split_video
4.11%
(0.11%)
4×
classify_videos:8:classify_frame
82.56%
(0.01%)
1000×
api_simulator:7:setup_api
13.00%
(0.00%)
4×
3.99%
1004×
65.46%
1000×
16.94%
1000×
13.00%
4×
~:0:<method 'read' of 'cv2.VideoCapture' objects>
3.99%
(3.99%)
1004×
api_simulator:11:get_api_prediction
65.46%
(0.03%)
1000×
image_utils:16:remove_edge_fast
16.94%
(0.04%)
1000×

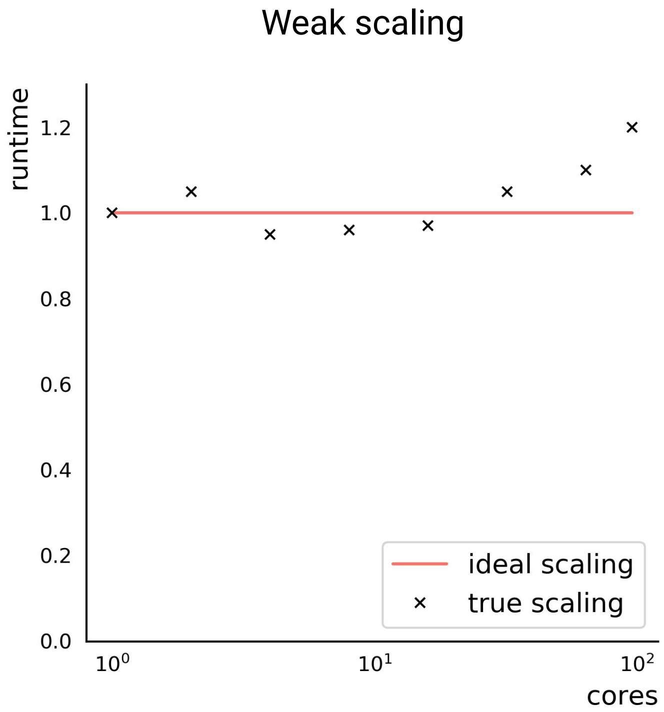
Measuring scaling

- How much resources can be used efficiently?
- How does run time change with more resources
  - *Weak scaling*: more resources and larger setup
    E.g. 5x as many cores and videos; same runtime?
  - *Strong scaling*: more resources and same setup
    E.g. 5x as many cores, same number of videos; 5x speedup?
- No program strong-scales indefinitely



Measuring scaling





Profiling with threading

- Not all profilers deal well with threading
  (e.g. `cProfile` will show all time in `join()`)
- Some deterministic profilers can do it (e.g. `yappi`)
- Easier for statistical profilers
  - Which function/line is being executed?
  - Which function holds the GIL
- Pyflame has two modes:
  - Measure GIL-holding functions only
  - Measure real time on each thread (`--threads`)

Demo:
profiling with threading

Demo notes:

- Example: work and sleep (threaded)
- Try profiling with cPython, see what goes wrong
- Use pyflame:
  - Now join() doesn't take time.
  - sleep() takes almost no time
  - All time taken by work(), it holds the GIL
  - Now use --threads option. Get 2 profiles: one for each thread.

Profiling with multiprocessing

- Explicit support is rare in native python
- Options:
  - Modify code to collect profile on each process:
    e.g. `cProfile.run('do_stuff', profile_file_name)`
  - Attach sampling profiler to process of choice
    (careful: the process must have representative workload)

Summary

1

Measure, don’t guess.

2

Profile before optimising

(this includes parallelising)

3
Simple profiling is easy

Thank you

54 Welbeck St, Marylebone,
London W1G 9XS, UK

Follow us: