Make your code more efficient (part 1)

Hadrien Grasland

2024-03-29

Disclaimer

- This is an **introductory** course
  - You can’t become an expert in 2 mornings
  - But you can learn the general process + simple know-how
  - Open to adding advanced courses: suggest topics!

Why optimize programs ?

- Put computing resources* to better use
  - Solve the same problem with less resources
  - Solve more/bigger problems with the same resources
- Be nice to people (users, other developers, yourself)
  - Here, key metric is interaction → output delay
  - Frequent waiting feels unpleasant, breaks focus

* Not just about using hardware X for time T : resource-associated costs include buying, maintenance, power, cooling, environmental footprint…

Don’t count on hardware alone

- Hardware improvements **likely to slow down** soon
  - CPU/GPU transistor fins getting ~4nm wide as of 2024
  - Si lattice parameter is 0,5 nm → 2D scaling close to end
  - 3D stacking bad for heat dissipation
  - No *industrial-grade* replacement for Si FETs yet
  - Similar situation for other hardware

- Do you know how much you would need to wait/spend ?

Our optimization strategy

1. **Prepare for change**
2. Find the bottleneck
3. Study the state of the art
4. Improve the algorithm
5. Cater to hardware/OS needs
6. Know your programming language

Preparing for change

- Like all code changes, optimization is risky
  - May break normal functionality (wrong results!)
  - Today’s ideas may turn out to be useless/bad

- How do we prepare for this ?
  - **Version control :** Have a way back → Another course
  - **Tests :** Find out when you break things → Another course
  - **Benchmarks :** Have a metric for success → This course!

Benchmarks

- To speed things up, need to define slowness
  - Known **workload** that you want to use less resources
  - Resource usage **metrics** (e.g. execution time, RAM used…)
- Often, **execution time** is your starting point
  - Advantage: That's what you actually care about
  - Drawback: Most sensitive to HW/OS config, interference
  - Alternatives: Elapsed CPU cycles, bytes read/written…

The perfect benchmark

- **Easy to write, automated*, realistic:** Same as tests
- **Fast:** Usually only interested in one benchmark at a time
- **Precise:** Metric is measured quite precisely (±5% is easy**)
- **Reproducible:** Multiple runs provide comparable results

- Exhaustive fine-grained benchmark coverage is **not** needed
  - Start with slow real-world workload
  - Find component(s) responsible and benchmark these

* Automated benchmark *analysis* is hard, but aim for single-command *measurements*.
** If you keep unrelated system background load low while running benchmarks.

From macro to micro benchmarks

- Real-world workload may not be convenient to run
  - Long execution times
  - Big input data you can’t just commit in the repo
  - Accesses external resources (database, CVMFS…)
- **Micro-benchmarking** means making a simplified workload
  - Must still exercise the original source of slowness
  - Beware smart compilers, libraries, OS, etc. may use a different algorithm when processing simpler problems

Processing multiple things

- Common scenario: Processing N tasks gets slower as N grows
- Need to clarify our needs
  - Do we care about **latency** T(end, 1 task) – T(start, 1 task)?
  - …or only **throughput** Ntasks / (T(end, job) – T(start, job))?
  - Does resource usage grow **linearly** with N?
- Good idea to explore with exponential 2^n input sizes
- Run benchmarks long enough to amortize transients*

* OS process startup overheads, CPU frequency scaling, CPU and disk cache warm-up…
1s typically sufficient for CPU- or memory-bound work without initialization phase.

Practical: Microbenchmarking

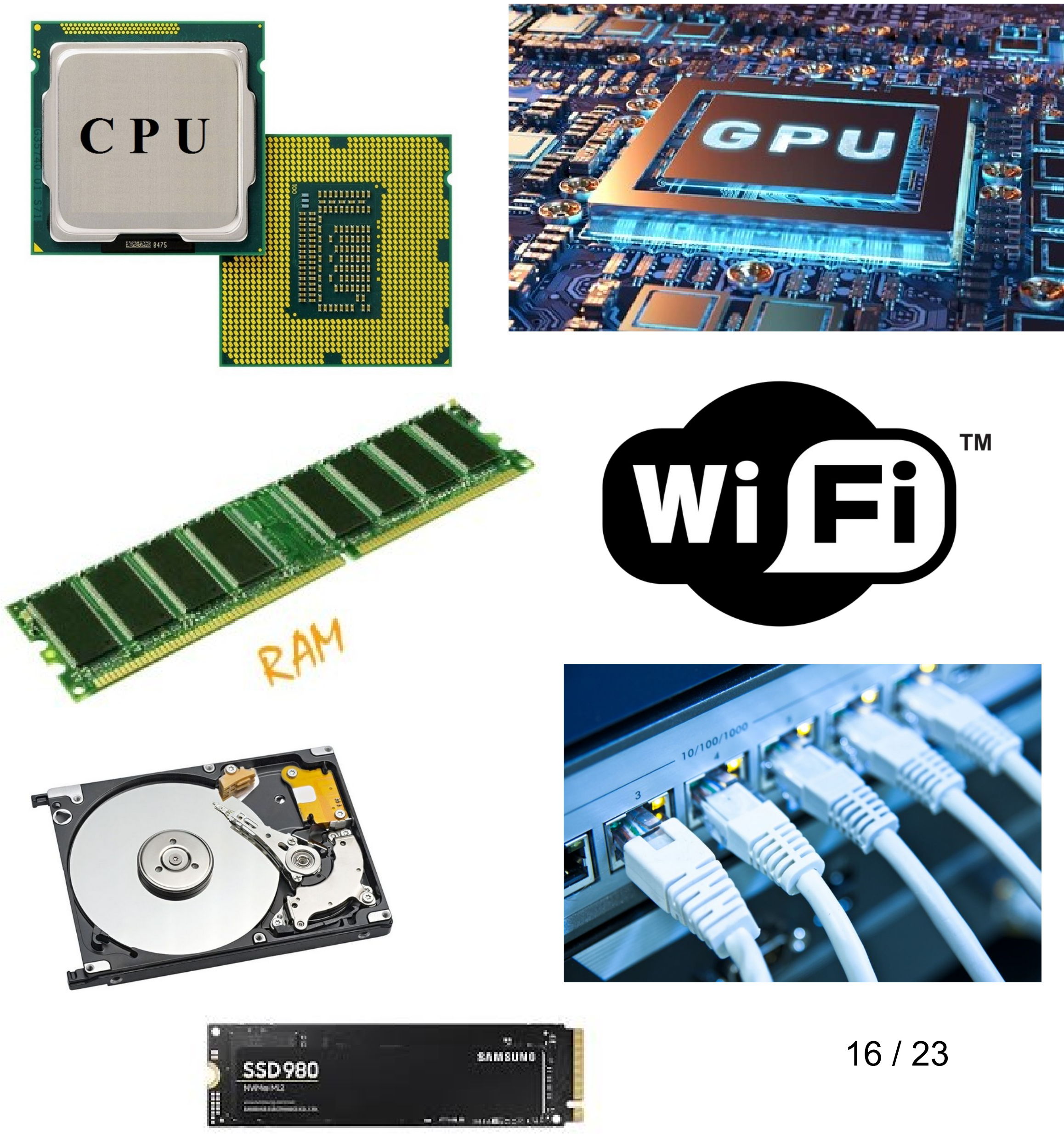
https://grasland.pages.in2p3.fr/make-your-code-more-efficient/microbenchmarking.html

Our optimization strategy

1. Prepare for change
2. **Find the bottleneck**
3. Study the state of the art
4. Improve the algorithm
5. Cater to hardware/OS needs
6. Know your programming language

Think about the whole system

- Computers let you access many resources
  - Hardware (CPU, RAM…)
  - Software (OS, database…)
- Each resource has limits
  - Some of these limit your code’s performance
  - Need to find which ones!



The USE method

- Enumerate system resources your program may use*
  - Internal components, external services, interconnects...
- For each resource, check...
  - **Utilization** (relative/absolute time spent servicing requests)
  - **Saturation** (queued work that can’t be serviced yet)
  - **Errors** (problems servicing requests)
- More from inventor: https://www.brendangregg.com/usemethod.html

* This step can be difficult for complex programs, you may want to call your local expert.

USE method advice

- Remember to check **individual CPU cores, disks**…
    - Your program may not use all of them yet
- Think about **interconnects** (CPU-RAM, CPU-GPU, network…)
- Think about the **outside world** (shared storage, database…)
- With VMs, containers, multi-user systems…, also check **host metrics** and **user quotas** (call admins for help!)
- Utilization >70% may already indicate a bottleneck

Metrics all the way down

- Complex resources provide finer-grained usage metrics
- Using CPU as an example, can measure among other things…
  - **Clock rate** (should be ≥ base clock for CPU-bound code)
  - **Instructions per cycle** aka IPC (should *usually* be ≥ 2)
  - **Cache hit/miss** at L1, L2, L3 + **RAM bandwidth**
  - Number of **branches**, rate of **misprediction**
- Requires more expertise*, but provides very valuable insight!

* Actually the topic of a whole other course.

Profiling

- You found a bottleneck! Narrow down which code faces it
  - **Process monitor:** Which processes use most CPU time?
  - **CPU profiler:** Within a process, which code uses most CPU?
  - **Memory profiler:** Suspicious allocation/liberation patterns?
  - **Storage:** Check out syscalls, kernel block device metrics
  - **Network:** Break down traffic per connection
- Beware: Fine-grained tools are specialized for one resource
  - Make sure that resource truly is your bottleneck!

What if I can’t find the right tool?

- Use the performance equivalent of printf debugging!
  - Check elapsed time in each function called by main()
  - Recursively apply this method in functions using most time
- Beware of **clock pitfalls**
  - Use fine-grained clocks (~ns for Linux monotonic clock*)
  - Checking time is not free (~40ns on Linux)
  - Expect small-scale outliers (~µs spent on OS interrupts)
  - Checking clock in a loop can prevent loop optimizations

* The OS API behind C++’s std::chrono::steady_clock and Python’s time.perf_counter()

Practical: Finding the bottleneck

https://grasland.pages.in2p3.fr/make-your-code-more-efficient/find-the-bottleneck.html

The story so far

- Optimize to put resources to better use, be nice to people
  - …or when you can't just throw more HW at the problem
- Prerequisites for effective performance optimization
  - Have an **easy way back** when you do it wrong
  - Make sure you will **notice breakage** early on
  - Set a reproducible **benchmark** + associated metric
  - **Narrow down** which code needs most care and why

Study the state of the art

- Did someone else solve the same problem before?
  - **Standard library** of your programming language
  - Common **utility libraries** (FFTW, BLAS/NumPy, HDF5…)
  - Domain-specific external packages
  - Computing publications, blogs, StackOverflow…
- Try their solution, measure if it performs better!
  - If code can’t be reused as-is, study the algorithms and data structures

Day 1 wrap-up

- Now we’re ready to optimize our code
  - We know which code needs care, and why
  - We can confidently change it + assess outcome
  - We have asserted we’re not reinventing the wheel
- Day 2 of the training will introduce how we optimize
- **Homework:**
  - Finish exercises from previous practicals, ask questions
  - Find the bottleneck of this program

Thanks for your attention!