Make your code more efficient (part 1)

Hadrien Grasland

2026-04-17

Disclaimer

- This is an **introductory** course
  - Won’t become an expert in 1 day
  - But can learn the general process + simple know-how

Why does speed matter?

- Put computing resources* to better use
  - Solve same problem using 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, environment footprint…

More hardware (alone) won't help

- **Serial code** at best ~10% faster per HW generation (~2 years)
  - Less lucky code doesn't improve, may slow down
- **Parallel code** doesn't scale linearly with core count
  - Clock rates shrink, sync. costs grow, Amdhal hates you
- **New hardware units** focused on neural nets, not numerics
  - Can only be used via specially crated code
  - Awfully low precision, only fast for matrix products
- **How much** can you wait/spend?
  - See hardware supply woes, exploding prices & watts…

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 stuff → Another course
  - **Benchmarks :** Have a metric for success → This course!

Benchmarks

- To get faster, must measure speed
  - Known **workload** that should use less resources
  - Resource usage **metrics** (e.g. execution time, RAM used…)
- Often, **execution time** will be your starting point
  - Advantage: That’s what you usually care about
  - Drawback: Sensitive to HW/OS config & interference
  - Alternatives: Elapsed CPU cycles, bytes read/written…

The perfect benchmark

- **Easy to write, automated*, realistic:** Just like tests
- **Fast:** Individual benchmarks take a few seconds to run
- **Precise:** Keep background noise/bias low (±5% is easy**)
- **Reproducible:** Same result after e.g. machine reboot
- Full fine-grained benchmark coverage is **not** needed
  - Start with a slow real-world workload
  - Find component(s) responsible, benchmark these

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

From macro to micro benchmarks

- Real-world workload may not be easy to benchmark
  - Long execution times
  - Big input you can’t put in your code repository
  - Uses external resources (database, CVMFS…)
- **Micro-benchmarking** means simplifying. Do it with care:
  - Must still exercise original source of slowness
  - Smart compilers, libraries, OSes… may use a different algorithm when processing simpler problems

Problem size effects

- Speed issues often appear in loops over N inputs/tasks
- Then must clarify our need
  - Do we care about **latency** T(end, 1 task) – T(start, 1 task)?
  - …or only **throughput** Ntasks / (T(end, job) – T(start, job))?
- Exponential problem sizes (e.g. 2^n) good for exploration
  - Execution time may not grow linearly with N
- 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

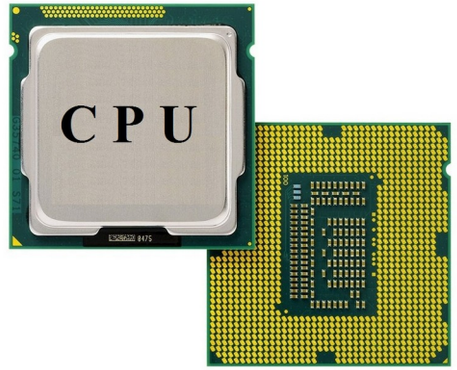
https://informatique-des-deux-infinis.pages.in2p3.fr/pheniics/make-your-code-more-efficient/microbenchmarking.html

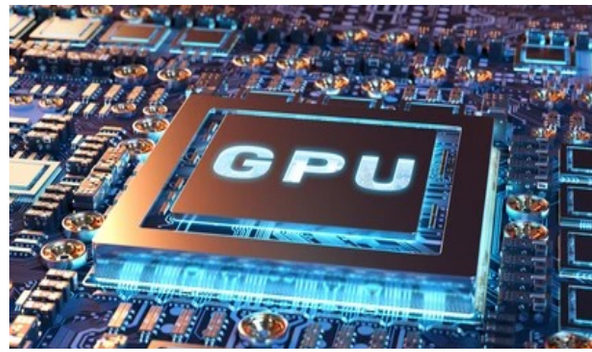
Our optimization strategy

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 (web, database…)
- Each resource has limits
  - Some of these limit your code's performance
  - Need to find which!





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 author: https://www.brendangregg.com/usemethod.html

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

USE in practice

- Remember to check **individual CPU cores, disks**…
  - Your program may not be able to use all of them
- 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!)
- Problems often start before 100% utilization

Demo: Find the bottleneck

https://informatique-des-deux-infinis.pages.in2p3.fr/pheniics/make-your-code-more-efficient/find-the-bottleneck.html

Metrics all the way down

- Complex resources have finer-grained usage metrics
- Using CPU as an example, can measure things like…
  - **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 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 syscalls, block device metrics…
  - **Network:** Break down traffic per connection
- Beware: Fine-grained tools are specialized for one resource
  - Make sure that resource is your bottleneck!

Can’t find the right tool?

- We have printf debugging too!
  - Check elapsed time in each function called by main()
  - Recursively apply to functions using most time
- …but beware: **clocks have pitfalls**
  - Use fine-grained clocks (~ns for Linux monotonic clock*)
  - Checking time is not free (~40ns on Linux)
  - Outliers at small scales (~µs spent on OS interrupts)
  - Checking clock in a loop hampers loop optimizations

* Used by C++’s std::chrono::steady_clock and Python’s time.perf_counter()

Practical: CPU profiling

https://informatique-des-deux-infinis.pages.in2p3.fr/pheniics/make-your-code-more-efficient/profiling.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 the most care, why

Final preparation: 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, at least study algorithms and data structures

Morning wrap-up

- We are now ready to optimize our code
  - We know which code needs care, and why
  - We can confidently change it, assess outcome
  - We have made sure we're not reinventing the wheel
- This afternoon, we'll see how to actually make it faster!

Thanks for your attention!