Make your code more efficient (part 2)

Hadrien Grasland

2026-04-17

Part 1 reminder

- Preparations before code optimization
  - Set up **version control** (if you're not using it already)
  - Write more, **finer-grained tests**
  - Define **benchmark**: workloads + metrics
  - Find the **system bottleneck** + what code stresses it
  - Check out the **state of the art** for this problem

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

Performance mantras*

1. **Don’t do it:** Do you need to do this at all (e.g. debug leftovers)?
2. **Do it, but don’t do it again:** Can you keep/reuse the result?
3. **Do it less:** Can you do it every N inputs, only in debug mode…?
4. **Do it later:** Can you e.g. amortize fixed costs by batching?
5. **Do it when they’re not looking:** Think about human wait time!
6. **Do it concurrently:** Modern computers can do parallel work
7. **Do it cheaper:** Most of today’s lecture!

* Stolen from Brendan Gregg’s beautiful collection of performance checklists.

Practical examples

- **Memory allocation**
  - ns → µs scale: Not that expensive, but avoid in tight loops
  - General idea: Reset and reuse previously created objects
- **File I/O and console printouts**
  - Do you need to print/save/load all this data?
  - Can you live with a subset of data sometimes? Always?
  - Can you reduce the precision of stored data at some point?
    - Computations often need more precision than storage

Algorithm complexity primer

- Often, code has trouble scaling up to **larger datasets**
  - Performs fine at small scale, too slow at large scale
- Standard approach when facing this kind of issue
  - Find one or more problem size metrics N, M...
  - Determine how compute time scales with these
  - Assume large problem size → Neglect low-order terms
  - e.g. linear search for M things in a list of size N is O(N*M)

What algorithm complexity tells us

- **O(1):** Problem size doesn't matter (e.g. querying array length)
- **O(log(N)):** Exploited input structure (e.g. binary search)
- **O(N):** Standard complexity if you need to use all inputs
- **O(N*log(N)):** Difference with O(N) rarely matters
- **O(N²):** Will struggle with larger problem sizes
- **O(N^3), O(2^N), O(N!), etc.:** *Unusable* at large problem sizes

- Of course, sometimes you don't have a choice
  (e.g. can't multiply NxN matrices in O(N²) time)

Limits of algorithm complexity

- Assumes **asymptotically large problem size**
  - Low-order terms may be important at your problem size
  - High-order terms may not matter so much
- Does not express many important algorithmic features
  - Constant resource usage **multipliers**
  - **Early exit** optimizations (e.g. filter early, strongest filter first)
  - **Threshold effects** (e.g. running out of CPU cache)
  - Code complexity and **maintainability**

Example: List search

- Searching something in a list of N elements can be…
  - O(N) with **linear search** (look up each element in order)
  - O(log(N)) with **binary search** (sort items by search key)
  - O(1) with **hashing** (derive array index from search key)
- …but there are **other implications**
  - Modifying sorted “lists” is hard/slow (need special trees)
  - Hashing can be more expensive than comparison
  - Varying key requirements + different data structures

Practical: Basic optimizations

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

Why do we care ?

- Hardware performance characteristics are not homogeneous
  - **Latency** improves more slowly than **throughput**
  - **Memory hierarchy:** slow and large vs fast and small
  - Some HW/OS features only available via **special APIs** (e.g. asynchronous I/O, madvise, GPU…)
  - **Shared resources** slower/less predictable than private ones
- Big gain if bottleneck becomes something HW does well!

Low-level topics

- Could talk about DRAM, CPU, storage, network, GPU…
  - Not enough time: will focus on x86* CPUs and DRAM speed
- Will more specifically focus on **memory access**
  - Common issue, one of few things you control in Python
  - See μarch slides this part comes from for advanced topics

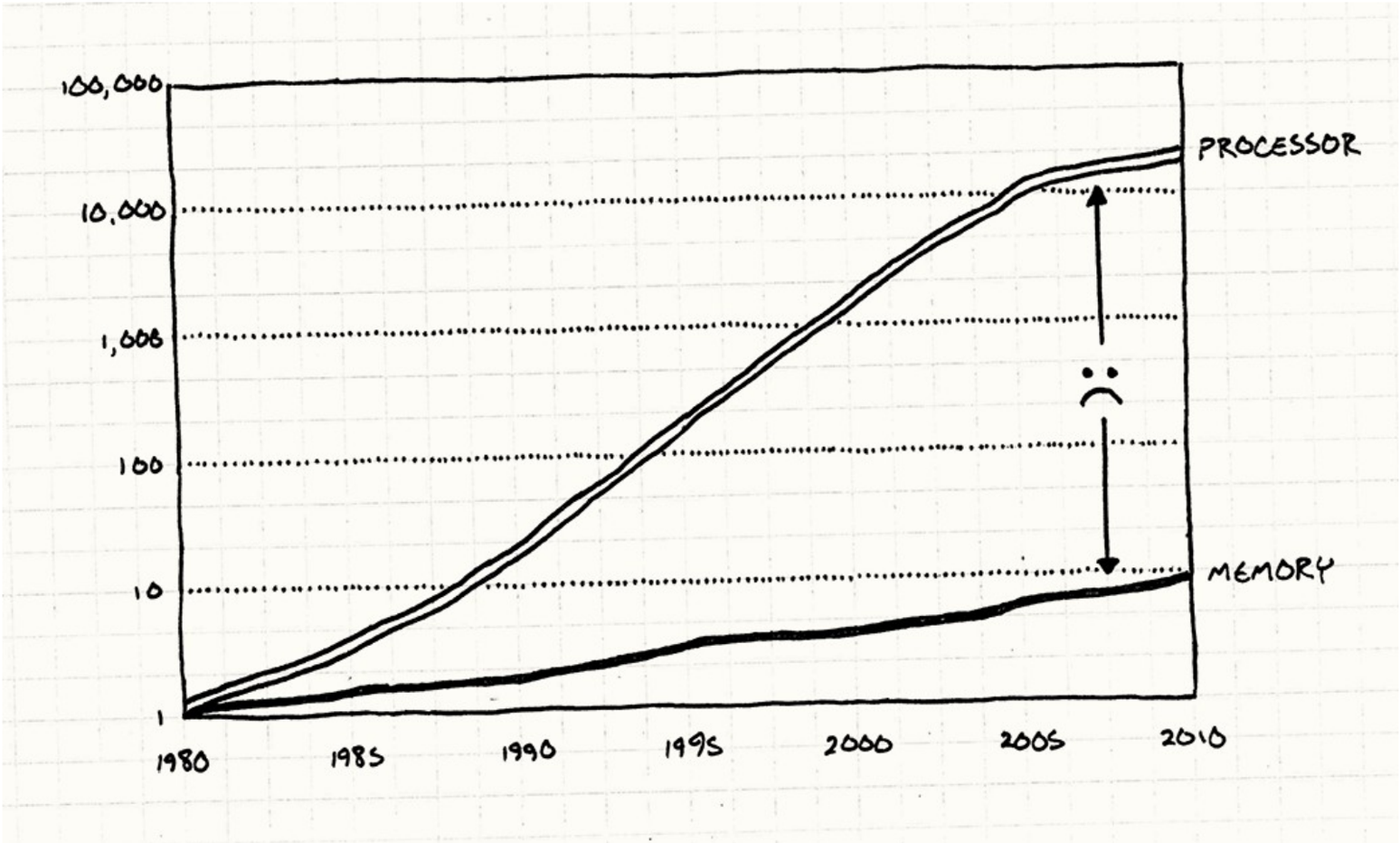
* Any CPU that inherits design from the Intel 8086, i.e. all current Intel and AMD CPUs.

Compute vs RAM performance

- Let's take a 2x AMD EPYC 7702 compute server from my lab
- How fast can it handle single-precision floats?
  - **Float arithmetic: 8.2 x 10^{12} f32 operations/sec**
    (2 sockets x 64 cores x 2 GHz x 2 FMA/cy x 16 operations/FMA)
  - **CPU-RAM bandwidth: 1.0 x 10^{11} f32 transfers/sec**
    (2 sockets x 204,8 GB/sec ÷ 4 B/f32)
- Meaning: On this CPU, below **~80 operations/RAM transfer***, CPU-RAM bandwidth is the performance limit!

* This metric, called "arithmetic intensity", can be used to guide optimization priorities.

…and it keeps getting worse*



http://gameprogrammingpatterns.com/data-locality.html

* Longer discussion by BLAS/LAPACK author: https://www.youtube.com/watch?v=cSO0Tc2w5Dg

Can I have faster memory?

- Computer storage balances many concerns
  - **(Non-)Volatility** (preserve data when off)
  - **Performance** (latency, bandwidth)
  - **Energy** efficiency
  - **Density**
  - **Price**

- A **memory hierarchy** gives us a bit of everything

Memory hierarchy example

- Our 2x AMD EPYC 7702 system contains many memories:
  - Regs : **cap.** 16x32B **lat.** 0cy **BW** ~unlimited
  - L1d : 32 KiB 4-8cy (2+1)x32B/core/cy
  - L2 : 512 KiB ≥ 12cy 32B/core/cy
  - L3 : 4 MiB/core ~39cy 32B/core/cy
  - RAM : ~2 GiB/core ~10^2cy 1,6B/core/cy*
  - SSD : ~1 TiB/**node** ~10^5cy 0,02B/core/cy**
  - HDD : ~10 TiB/node ~10^7cy 0,001B/core/cy**
  - NFS : See SSD/HDD + network limits + inter-node sharing

Non volatile

* Each socket has 8 DDR4-3200 channels (204,8 GB/s), shared by 64 cores running at 2 GHz
** Assuming ~100µs lat, 4 GB/s BW for SSDs and ~10ms latency, ~200MB/s BW for HDDs

Know your caches

- **Spatial locality:** Caches work with aligned “lines” (~64 bytes)
  → Group data by usage, arrays of structs require care
- **Temporal locality:** New lines replace least-recently-used ones
  → Don’t wait too much before reusing data
- Aim for **exhaustive/linear access patterns**
  → Otherwise issues w/ HW prefetch, associativity, TLB…
  → Can reduce damage with padding, manual prefetch…
- **Beware shared writes** in parallel code (~RWlock per line)

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**

Python

- Easy to get started with, tons of libraries available
- But official implementation (CPython) slow to execute code
- Other impls face lang design issues, low library/tool support
- Low hardware control → Less optimization headroom

- Consequences:
  - Most of your effort should be spent studying library docs
  - Programs should be bottlenecked by long-lasting calls to libraries written in another language (C/++, Fortran, …)

C++

- More low-level control, compiler optimizes a lot more
- But by the creator's admission, no one fully understands it
  - Everyone has their "good part". That doesn't work in teams.
- Dependency management is a pain → Lower code reuse

- Consequences:
  - Learning it is a big investment, may not pay off
  - Better for exotic problems with few/no/poor existing libs

C++-specific: Know your compiler

- By default, most compilers don’t build for max performance
- GCC/clang options you should be familiar with:
  - **O3**: Optimize as much as allowed by other rules
  - **march=xyz**: Build for CPU xyz, not every CPU since 2003
    - **-march=native**: Build to run on the same machine
  - **ffast-math**: Treat floats as real numbers (dangerous)
    - Use it to find potential optimizations, don’t leave it on

Other options?

- **Beware!** How will you convince your team it's a good idea?
- But for personal enlightenment, try learning…
  - **Julia:** High-level like Python, different perf tradeoffs
  - **Rust:** Rebuilding C++ with 20+ years of hindsight
  - **Fortran*:** If array compute is all you need, it's very good at it
- Languages are just the beginning, mastering **big libraries** (numpy, SYCL…) is a lot of work too!

* I am talking about modern Fortan here (>= 90), which is quite different from the Fortran <=77 that you'll find in dusty numerical codebases.

Practical: Advanced optimizations

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

Conclusion

- **Prepare** before you optimize :
  - Version control, testing, benchmarking, profiling
  - Find perf-critical problem, study state of the art for it
- Then **speed up** that bottleneck:
  - Start with human/algorithm intelligence for max benefits
  - Then sync up with hardware needs for the last factors
- Programming languages are all about **compromises**.
  - Pick the right tool for the right job.

Further reading

- For Linux performance analysis, get to know Brendan Gregg
  - His website is messy but full of hidden treasures
  - Systems Performance book is a *great* intro + reference
  - BPF book more advanced, many tools need root access
- As for compute performance optimization…
  - Many specialized books, few general ones / good first picks
  - Denis Bakhvalov's book would be my recommendation
- **Recommended books are available at the IJCLab library!**

Need a more advanced course?

- This summer, colleagues organize the Gray Scott School
  - 2 intense weeks on many numerical computing topics
  - Covers CPU and GPU + many languages, libraries & tools
- Three ways to join:
  - Face-to-face at LAPP, Annecy
  - Via participating computing center
  - Watch live stream “à la carte”
- Register using this form

Thanks for your attention