Make your code more efficient (part 2)

Hadrien Grasland

2024-04-05

Day 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 + associated metrics
  - Find the **bottleneck**, and what code is limited by it
  - Check out the **state of the art** for this problem

Homework wrap-up

- The limiting step of this program is **searching entries in a list**
  - For each queried element, the whole list is searched
- Simplified model : search for occurences of M elements in a list of length N, it takes time T to examine one list element
  - Time to search for one element : N * T
  - Time to search for all elements : M * N * T
- We can do better than this by using **better algorithms**

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 really need to do this?
2. **Do it, but don’t do it again:** Can you keep/reuse the result?
3. **Do it less:** Can you do it e.g. only during debugging?
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:** Remember computers can do parallel work
7. **Do it cheaper:** Most of today’s lecture!

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

Example areas of application

- **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 it most of the time? Always?
  - Can you reduce the precision of stored data at some point?
    - Compute precision does not have to be the same!

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)
- **$O(\log(N))$:** It doesn't have a big impact (e.g. binary search)
- **$O(N)$:** Standard complexity if you need to use all inputs
- **$O(N*\log(N))$:** Difference with $O(N)$ usually doesn't matter
- **$O(N^2)$:** Major slowdown at larger problem sizes
- **$O(N^3)$, $O(2^N)$, $O(N!)$, etc.:** Painful at large problem sizes

- Of course, sometimes you don't have a choice
  (e.g. can't multiply NxN matrices in $O(N^2)$ 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 elements by search key)
  - O(1) with **hashing** (derive array index from search key)
- …but there are **other implications**
  - If element list varies, need trees for sorting (slower)
  - Hashing can be a lot more expensive than comparison
  - Varying key requirements + different data structures

Practical: Algorithmic optimizations

https://grasland.pages.in2p3.fr/make-your-code-more-efficient/algorithmic-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 ?

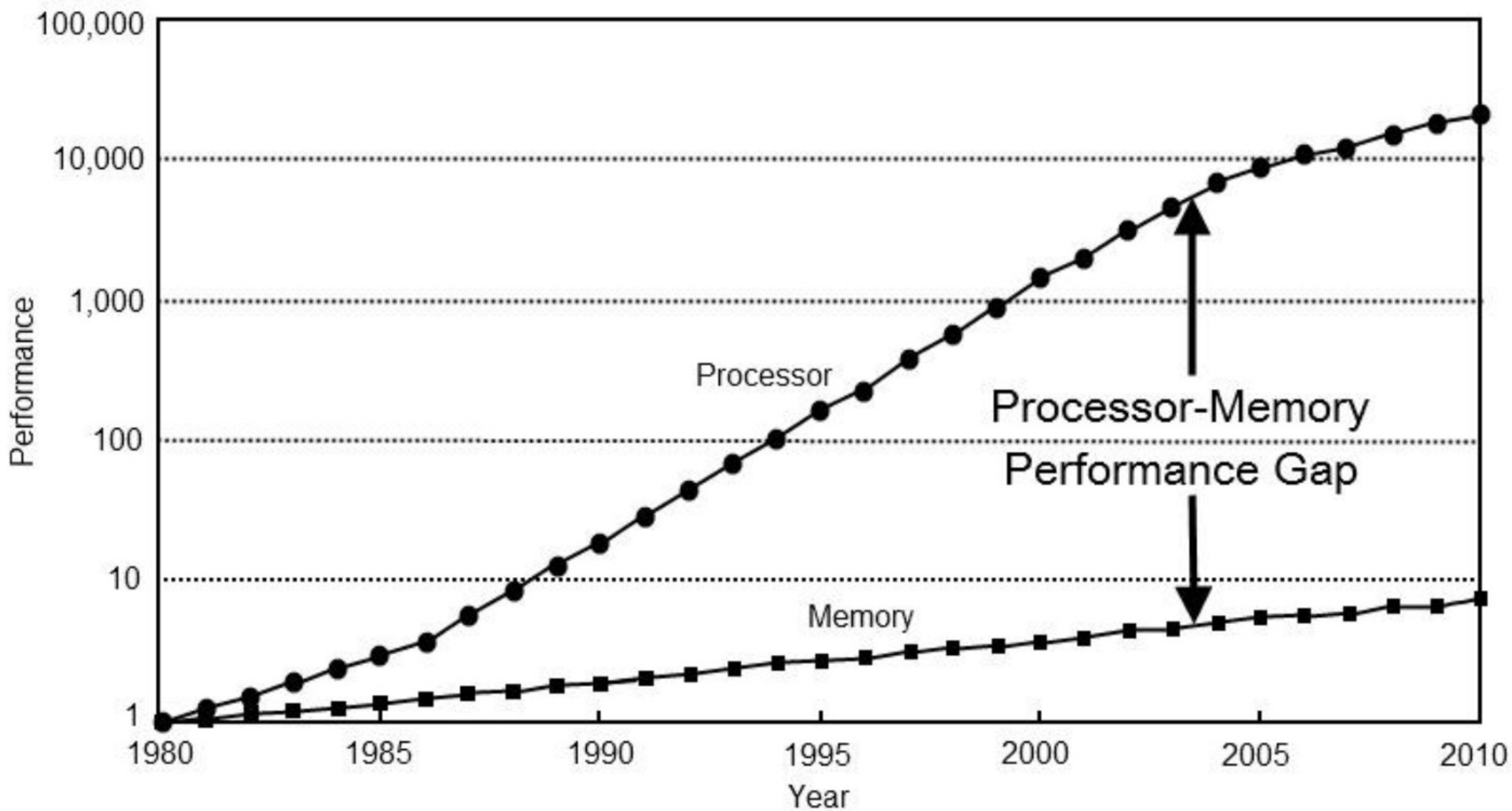
- Hardware performance characteristics are not homogeneous
  - **Latency** improves much more slowly than **throughput**
  - **Memory hierarchy:** slow and large vs fast and small
  - Some HW/OS features only available via **weird system APIs** (e.g. asynchronous I/O, madvise, GPU…)
  - **Shared resources**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

Scope

- Could talk about DRAM, CPU, storage, network, GPU…
  - Not enough time: will focus on x86* CPUs and DRAM speed
- Will discuss…
  - Memory optimizations
  - Logic optimizations
  - Arithmetic optimization
  - Vectorization
  - Multithreading

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

The memory wall



Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

Some numbers

- atlas1.ijclab.in2p3.fr: A modern/fast compute node with two AMD EPYC 7702 64-core CPUs
  - **Compute throughput:** 2 sockets x 64 cores x 2 GHz x 2 FMA/cy x 16 f32 ops*/FMA = **8.2 x 10^{12} f32 ops / second**
  - **DRAM bandwidth:** 2 sockets x 204,8 GB/s = 409,6 GB/s = **1.0 x 10^{11} f32 transferred / second**
- **Interpretation:** For each f32 you read from DRAM, if you're not doing 80 f32 computations, you're limited by memory speed.

* Multiplication or addition, by convention. So FMA (fused a*b+c) counts as 2 operations.

Avoiding the memory wall

- CPUs provide small, fast chip-local memories called **caches**. Keeping the example of AMD EPYC 7702, each socket has...
  - 256 MB L3, shared bw cores, latency 39 cy, bwidth 32B/cy
  - 512 KB/core L2, latency 12 cy, bandwidth 32B/cy
  - 32 KB/core L1i+L1d, specialized for code/data, latency 4-8 cy, bandwidth 2x32B/cy read + 32B/cy write
- As long as most of your data fits in L1d cache, you can get away with doing only **one computation per memory load**!

CPU cache properties

- **Automatic:** Every memory read or write gets through caches
- **Granularity:** Even if you ask for 1 byte, CPU will get 64 bytes*
  - Data used together should be at neighbouring addresses
- **LRU policy:** Old data is evicted to make room for new data
  - Reuse previously loaded data as soon as possible
- **Beware large strides** (accesses to widely spaced addresses): Cause trouble with associativity, TLB, 4K aliasing…

* This number is x86 specific and could change someday.

Latency hiding

- Even the L1d cache has a few cycles of latency
- CPUs try to handle this by processing N instructions in parallel
- This nice plan may be foiled in various situations:
  - You rely excessively on **high latency** caches, DRAM
  - You have lots of **indirections** (e.g. arrays of pointers)
  - More generally, you have **long dependency chains** (each instruction uses the result of the previous instruction)
  - Lots of **branches** (if/else, switch, …) with irregular conditions

Dependency chains in practice

- More of a concern with C++, Numba, … not with CPython
- If, you need to, say, sum a bunch of floats, avoid this pattern:

```
float acc = 0.0;
for (size_t i = 0; i < N; ++i) acc += input[i];
```

- Prefer something like this* (assuming M divides N):

```
std::array<float, M> accs { 0.0, 0.0, …, 0.0 };
for (size_t i = 0; i < N; i += M) {
  for (size_t j = 0; j < M; ++j) accs[j] += input[i+j];
}
// ...and then sum accs...
```

* If you use STL algorithms, this is how std::reduce *tries* to differ from std::accumulate

Logic optimization

- **Conditionals** (if, switch, etc.) are not free
  - CPU can only process 1/cy, can do most other ops 2+/cy
  - Condition must be predictable, failure is costly (15-20cy*)
  - Use them sparingly in loop + group by condition if you can
  - Consider “branchless” techniques if all else fails
- **Virtual methods** (from C++ OOP) can be costly
  - Prevent inlining → More latency, more instructions…
  - Fine in high-level code, ban them from tight compute loops

* As measured on the relatively old Haswell architecture, may have changed a bit since.

Arithmetic optimization

- Floating-point ops aren't born equal. Measured throughputs*:
  - ADD, SUB, MUL, FMA: 2 ops/cycle
  - DIV, SQRT: 0.25-0.33 ops/cy (6-8x slower)
  - EXP, LOG: 0.17-0.2 ops/cy (10-12x slower)
  - SIN, COS: 0.09-0,1 ops/cy (20-22x slower)
  - ATAN: 0.05 ops/cy (44x slower)
- **Consequences:** Keep it simple, reuse inverses, and prefer trigonometric identities over computing sin(atan2(x, y))

* Measured on 2015 hardware, the situation may have evolved a bit since then.

Should compilers optimize floats?

- **FP numbers are not real numbers** e.g. (a + b) + c != a + (b + c)
- Any operation reordering changes roundings, and thus results
  - Already a problem if you rely on strict equality for validation
- Some reorderings are unsafe (overflow, underflow, cancelation)
  - Compilers may not have enough context to tell what is safe
- So unless you use special languages (e.g. Fortran)
  or compiler flags (e.g. GCC's -ffast-math), this is your job.

Vectorization

- Good news: **1 CPU instruction processes N numbers at once**
  - 2 x f64 or 4 x f32* with SSE (all modern x86 CPUs)
  - 4 x f64 or 8 x f32 with AVX (~80-90% of WLCG CPUs)
- Bad news: Code that uses it requires **a lot of work**
  - Only beneficial for “simple” operations (e.g. ADD, FMA, …)
  - You must do the same thing with each input
  - Very sensitive to how you lay out data in memory
  - Reconciling performance with HW portability is hard

* Notice how using simple precision, where appropriate, not only halves your memory bandwidth usage but also doubles your arithmetic throughput.

How to vectorize?

- Leave it to **someone else's library** whenever you can!
- If you must do it, prefer using a **hardware abstraction layer**
  - Compiler "vector extensions" from clang + GCC
  - Libraries: MIPP, xsimd, Vc, libsimdpp, someday std::simd…
- Other approaches that I would advise against:
  - Shape code so compiler autovectorizes it (hard, brittle)
  - Use SSE/AVX instructions directly (hard, not portable)

Multithreading

- In larger experiments, **you may not need to parallelize**
  - Running N independent jobs in parallel is very easy
  - If that's already an Nx speedup, you're done!
- Do it if you must **reduce latency or spare a shared resource**
  - CPU L3 cache, DRAM capacity & bandwidth…
  - Storage, network, …
- Beware: Hard to get right + make fast, especially in Python*

* In CPython, multiple threads cannot execute Python code simultaneously.
To work around this, you must use multiple processes, which makes communication hard.

Practical: Low-level optimizations

https://grasland.pages.in2p3.fr/make-your-code-more-efficient/low-level-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**

Python

- Easy to get started with
- Tons of libraries available, quite easy to adopt a new one
- But official implementation (CPython) slow to execute code
- Other impls face lang design issues, low library/tool support

- 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 or 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 old and dusty numerical codebases.

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.

Final homework : Image sharpening

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

Thanks for your attention!

If you need help with a perf problem, feel free to get back in touch!