# Applied Algorithms Lec 2: Optimization

Sam McCauley

October 21, 2021

Williams College

# Admin

- Office hours 3-5 today in TCL 306
- Do Assignment 0 if you haven't
- Assignment 1 released Wednesday; we'll talk about the assignment and handin instructions on Thursday

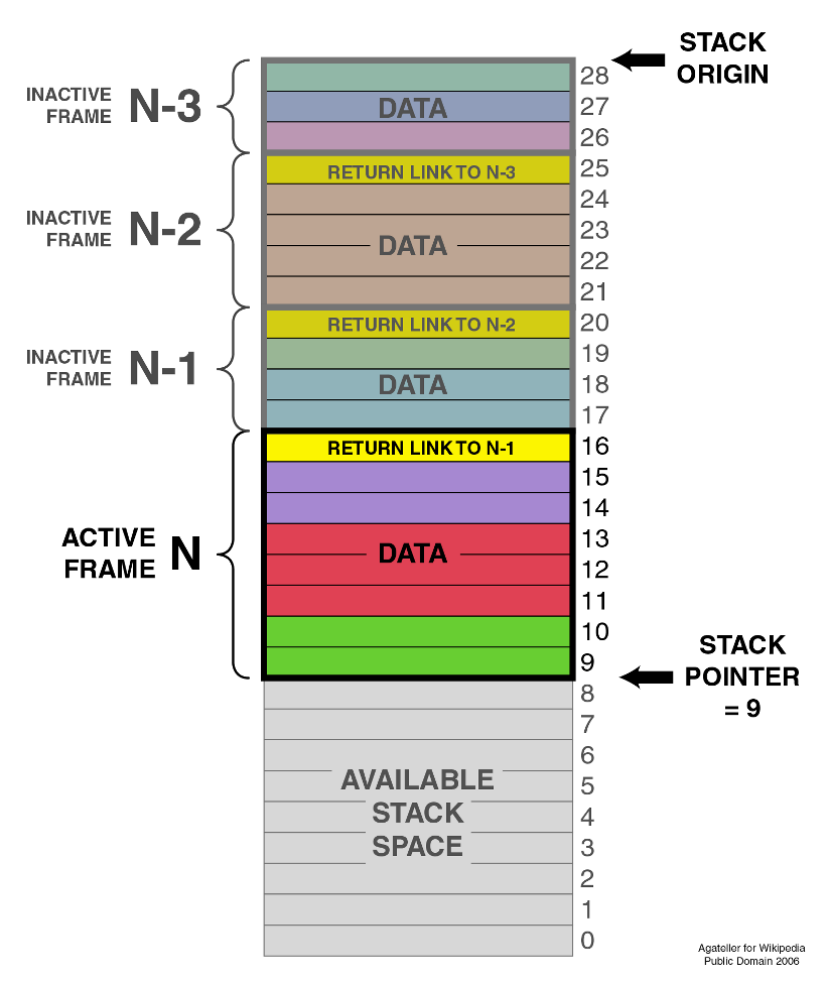
# Sorting in C

- `qsort()` from `stdlib.h`
- Takes as arguments array pointer, size of array, size of each element, and a comparison function
- What's a downside to this in terms of efficiency?
- Many ways to get better sorts in `C`:
  - Nicely-written homemade sort
  - `C++ boost` library
  - Third-party code
- Instructions to get this to work in handouts on the website (**strictly optional**)

# Architecture

- x86 architecture (not AMD, not M1)
- Intel i7-8700 "Coffee Lake" specifically
- This *is* likely to have an effect on fine-grained performance in some cases
- Your home computers are fine for correctness and coarse optimization; use lab computers for fine-grained optimization
- If I ask you to do a performance comparison, you should generally do it on lab computers. In any case you should write what you do it on.

# Where are things stored?




- In CPU register (never touching memory)
  - Temporary variables like loop indices
  - Compiler decides this
- Call stack
  - Small amount of dedicated memory to keep track of current function and *local* variables
  - Pop back to last function when done
  - **temporary**

# Other place to store things

- The heap!
- Very large amount of memory (basically all of RAM)
- Create space on heap using `malloc`
- Need `stdlib.h` to use `malloc`

# How to decide stack vs heap?

- Java rules work out well:
  - "objects" and arrays on the heap
  - Anything that needs to be around after the function is over should be on the heap
  - Otherwise declare primitive types and let the compiler work it out
  - Keep scope in mind!

# Makefile

- Each time we change a file, need to recompile that file
- Need to build output file (but don't need to recompile other unchanged files)
- Makefile does this automatically

# In this class

- I'll give you a makefile
- You don't need to change it unless you use multiple files or want to set compiler options
  - Probably don't need to use multiple files in this class
  - (Some exceptions for things like wrapper functions.)

# Let's look quickly at the default Makefile

- `make`, `make clean`, `make debug`

# Compiler flags

- `-g` for debug, `-c` for compile without build (creates `.o` file)
- Different optimization flags:
  - `-O1` or `-O2` are default levels
  - `-O3`, `-Ofast` is more aggressive; doesn't promise correctness in some corner cases
  - `-O0` doesn't optimize; `-Og` is no optimization for debugging
  - Other flags to specifically take advantage of certain compiler features (we'll come back to this)
- `-S` (along with `-fverbose-asm` for helpful info) to get assembly

# Variable types

- `int`, `long`, etc. not necessarily the same on different systems
  - On Windows `long` is probably 32 bits, on Mac and Unix it's probably 64 bits
  - `long long` is probably 64 bits
- Instead: include `stdint.h`, describe types explicitly
- Keep an eye out for unsigned vs signed.
- Quick example: `variabletypes.c`
- `printf` does expect primitive types

# Variable types cont.

- `int` (etc.) is OK for things like small loops
- If you care *at all* about size you should use the type explicitly
- Up to you when and where you use unsigned
  - Controversial in terms of style

# List of variable types

- `int64_t, int32_t`: signed integers of given size
- `uint64_t, uint8_t`: unsigned integers of given size
- `uint_fast64_t`: fastest int with at least 8 bits
- `uint_least8_t`: some unsigned int with at least 8 bits
- `INT64_MAX` (etc.): maximum value of an object of type `int64_t`

# Principles of Optimization

# Reminder

- "Premature optimization is the root of all evil!"
- Don't optimize your code until you have a working copy.
- Some gray area with structural decisions/trivial ideas—but until something works that is your main goal.

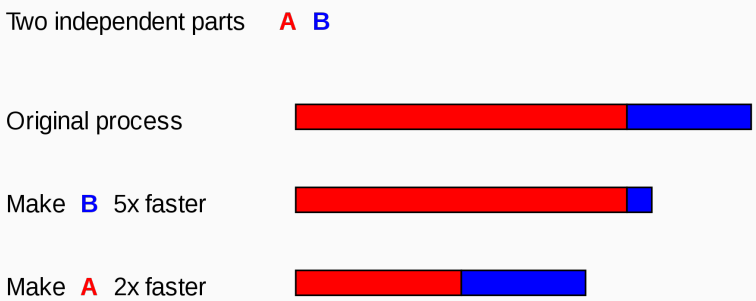
# Theory and Reality

- Computers are complicated! (And processors are proprietary!)
- Efficiency is always going to be highly experimental.
  - Sometimes something should work, but doesn't. Or vice versa.
- Goal for today: better understanding of where costs come from and how we can measure them

# Thought question

- What part of a program is most important to speed up?
- Let's say I have several functions. How can I choose which to optimize first?
- Answer: the one that takes the most *total* time
  - Time it takes $\times$ number of times it's called
  - May not be the slowest function—in fact, it's often a very fast but very frequently-used function
- Probably need to take into account potential to speed it up as well—I want the function that takes up the most time that I can save.

# Amdahl's Law




If a function takes up a $p$ fraction of the entire program's runtime, and you speed it up by a factor $s$, then the overall program speeds up by a factor

$$\frac{1}{1-p+p/s}$$

- Examples

# Amdahl's Law and Asymptotics

- Can *estimate* the total time of an algorithm asymptotically

- Example: Where to improve Dijkstra's algorithm?

# Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

# Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

# Measuring Performance

# What units to measure time?

- Overall: CPU time
  - Some idiosyncracies in how we're measuring it
  - CPU vs wall clock time shouldn't make much difference for us
- Costs of specific operations are generally given using number of "CPU cycles"
- Not-quite-accurate definition of a cycle: time to perform one basic operation

# Easiest way to measure time: just time it!

Easy, probably reflective of what you want. But some things to bear in mind:

- Make sure your timing is macroscopic.
  - No timing is exact.
  - CPU clocks usually only have a resolution of 1 million ticks per second (sometimes less)
  - Minimize issues with overhead, external factors
  - Rule of thumb: ideally an experiment will take 1 second
  - Always repeat several times and check consistency

# Timing one portion of your code

For Amdahl's, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

1. One option: factor out subroutine using separate testing code
    - Need to get info on how often it's called; simulate correct types of data.
    - Make sure the compiler does not optimize out your whole experiment!
2. Another option: Run same code with and without subroutine
    - Does that change the data the function is called with? Will the change in data affect running time?
3. Profiling!

We'll come back to this with some examples momentarily. Bear in mind: benchmarking itself is an entire area of computer science.

# Profiling code

- Why not just have your computer tell you what functions are caused the most, or keep track of how long they run, or monitor specific high-cost operations?
- Lots of such tools! We'll look at a couple of them right now, and use them throughout the class.
  - `gprof`
  - `cachegrind`
  - We won't use `perf` but some people like it
- What do you think some advantages and disadvantages are of using profiling software?

# gprof

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph
- *Quite* limited. But in some circumstances gives good advice.
  - Recursion; function-level resolution; cannot optimize; overhead; sampling problems

# `callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` - `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a virtual machine
- Gives information about costs you could not otherwise get, but VERY slow.

# Costs of Computation

# Note on time taken

Latency vs throughput:

- Latency: time it takes for a sequence of *data-dependent* operations of a given type
- Throughput: time after a previous operation when a new operation of the same type can begin.

Bear this distinction in mind when designing experiments!

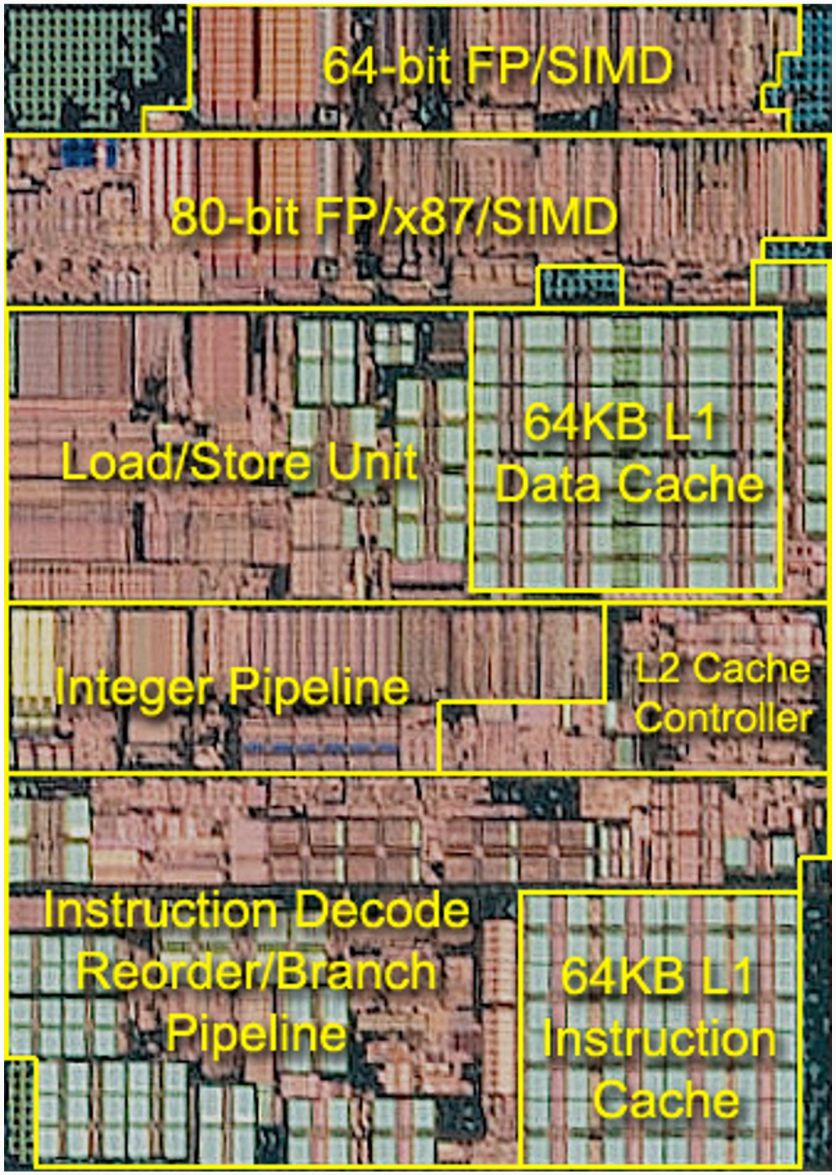
# Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
  - fast! 1-2 cycles
- Divide, modulo
  - Pretty slow; 5-20 cycles
- Float add, multiply?
  - Pretty fast on x86; almost as fast as integers

# More complicated operations

- Square root?
  - fast on our machines! 1-2 cycles
- memory allocation in bytes?
  - Pretty slow...
- memory allocation in megabytes?
- how does it grow as we increase the number of operations?
  - Cache efficiency is the problem here, not the memory call itself
  - (For what it's worth: `malloc` really is $O(1)$)

# Modern processors




- Lots going on
- Moving things around takes more time than processing

# Moving data around

- Casts can be expensive if they require moving the data into another part of the processor!

- (Can be free if they don't)

# Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?
- "Branch misprediction:" 10-20 cycles to fetch the new instructions from memory
- Can have similar issues with calling non-inlined functions (compiler is very good at avoiding this)

# Branch predictors

- CPU keeps track of your branches as it runs
  - Divides into four categories of how likely it is to be taken
- `gcc` also predicts your branches during compilation
- Can also give preprocessor directives about branches. Can be helpful (one of the last things you should do for optimization)

# Avoiding branch mispredictions

```
int max(int a, int b) {
   int diff = a - b;
   int dsgn = diff >> 31;
   return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
   a = a ^ b;
   b = a ^ b;
   a = a ^ b;
}
```

- Avoid branches (ifs, etc.)
- (Crazy tricks often not worth it nowadays)
- `cmov` operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Only way to be sure is to experiment

# Profilers examples: gprof

- Compile with `-pg` option; then run normally; then run gprof on the executable
- Gives information about what calls what and how much time is in each
- Not perfect, but gives us some information, especially for simpler programs
  - Can see if one function is called a LOT
  - Can see if one function is only ever called by one other function
- Gets confusing with recursive calls
- I may ask you to use this, but be aware that it's useful sometimes at best

# Profilers examples: cachegrind

- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind [your program]`
- Outputs number of cache misses for instructions, then data, then combined
- Simulates a simple cache (based on your machine) with separate L1 caches for instructions and data, and unified L2 and (if on machine) L3 caches
- Does L1 misses vs last level (L3) misses

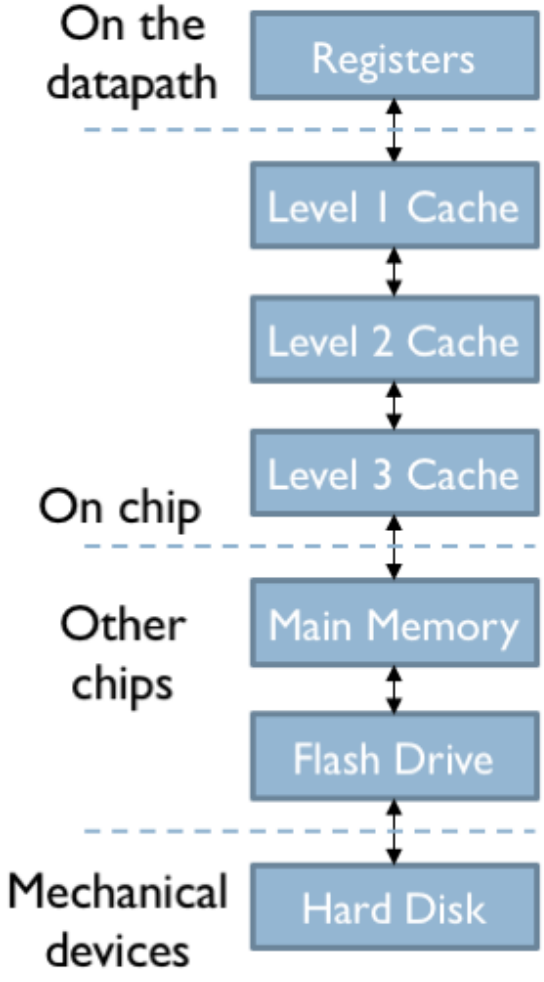
# Final major cost: cache misses!

- Data is stored in different places on the computer

- Cost to access it frequently dominates running time

# A Typical Memory Hierarchy

- Everything is a cache for something else…




| Access time | Capacity | Managed By |
|---|---|---|
| 1 cycle | 1 KB | Software/Compiler |
| 2-4 cycles | 32 KB | Hardware |
| 10 cycles | 256 KB | Hardware |
| 40 cycles | 10 MB | Hardware |
| 200 cycles | 10 GB | Software/OS |
| 10-100us | 100 GB | Software/OS |
| 10ms | 1 TB | Software/OS |

# How caches work

- Stores data in the optimal(ish) place
- Moves data around in *cache lines* of 64 bytes
- Modern caches are very complicated
- Can be advantages of adjacent cache lines
- Basically: close is good; jumping around is bad.

# Conclusions

- Different places where we can incur costs:
  - Operations
  - Branches and moving around instructions
  - Cache misses
- Determining costs is a matter of experimentation on modern machines!
  - Rarely perfect!
- Theme throughout class: design different experiments to test different aspects of code performance.