Writing Efficient Code in C++

Petr Ročkai

Organisation

- theory: ~20 minutes every week
- coding: all the remaining time
- passing the subject: collect 10 points
- most points come from assignments
- attendance is optional (but worth .1 point/seminar)

Assignments

- one assignment every 2 weeks, 5 in total
- you can get 3 points per assignment
- on my desk (in your repo) by 8am on Monday in 2 weeks
- you can be arbitrarily late, but will get 1 point less

Assignments (cont'd)

- you can use git, mercurial or darcs
- put everything that you want me to see on master
- write a simple `Makefile` (no cmake, autotools, …)
- each homework gets a target (`make hw1` through `hw5`)
- use the same repo for in-seminar work (`make ex1` …)

Competition

- we will hold a few (3 or 4) competitions in the seminar
- you'll have 40 minutes to do your best on a small problem
- the winner gets 1 point, second place gets .5 point
- all other working programs get .2 points
- we'll dissect the winning program together

Preliminary Plan

2.10.	cancelled (conference, sorry!)	
9.10.	microbenchmarking & statistics	hw01 due
16.10.	the memory hierarchy	
23.10.	using `callgrind`	hw02 due
30.10.	tuning for the compiler/optimiser	
6.11.	competition 1	hw03 due
13.11.	understanding the CPU	
20.11.	exploiting parallelism	hw04 due
27.11.	using `perf` + competition 2	
4.12.	Q&A, homework recap	hw05 due
11.12.	semester recap + competition 3	

Part 1: Introduction & Tools

Efficient Code

- computational complexity
- the memory hierarchy
- tuning for the compiler & optimiser
- understanding the CPU
- exploiting parallelism

Understanding Performance

- writing and evaluating benchmarks
- profiling with `callgrind`
- profiling with `perf`
- the law of diminishing returns

- premature optimisation is the root of all evil
- (but when is the right time?)

Tools

- `C++14` & a matching compiler (clang 3.7+, g++ 5+)
- on a POSIX operating system (preferably not in a VM)
- `perf` (Linux-only, sorry)
- `callgrind` (part of the `valgrind` suite)
- `kcachegrind` (for visualisation of `callgrind` logs)
- `gnuplot` for plotting performance data
- `brick-benchmark` for micro-benchmarks in C++

Part 2: Computational Complexity

Complexity and Efficiency

- this class is not about asymptotic behaviour
- but: you need to understand complexity to write good code
- performance and security implications

- what is your expected input size?
- complexity vs constants vs memory use

Quiz

- what's the asymptotic complexity of:
  - a bubble sort? (standard) quick sort?
  - inserting an element into a RB tree?
  - inserting an element into a hash table?
  - inserting an element into a sorted vector?
- what are the amortised complexities?
- how about expected (average)?
- what if the hash function is really bad?

Worst-Case Complexity Matters

- CVE-2011-4815, 4838, 4885, 2012-0880, …
- apps can become unusable with too many pictures/songs/…
- use a better algorithm if you can (or must)
- but: simplicity of code is worth a lot, too
- also take memory complexity and constants into account

Constants Matter

- n ops if each takes 1 second
- $n \log n$ ops if each takes .1 second
- n^2 ops if each takes .01 second

Picking the Right Approach

- where are the crossover points?
- what is my typical input size?
- is it worth picking an approach dynamically?
- what happens in pathological cases?

Exercise 1

- set up your repository and a `Makefile`
- implement a bounded priority buffer
  - holds at most n items
  - holds at most one copy of a given item
  - forgets the smallest item if full
  - fetch/remove the largest item
  - API: `insert`, `top` and `remove`
- two versions: sorted `std::vector` vs `std::set`

Exercise 1 (cont'd)

- write a few unit tests
- write a benchmark that inserts ($\sim 10^7$) random values
- the benchmark can use `clock(3)` or `time(1)`
- compare the approaches for $n = 5, 10, 10000$

- what are the theoretical complexities?
- what are your expectations on performance?
- can you think of a better overall solution?

Homework 1

- implement hash tables with insert and find
  - with linked-list buckets [1pt]
  - with linear probing and rehashing [1pt]
- compare with `std::set` and `std::unordered_set` [1pt]
- bonus: beat `std::unordered_set` by >10 % [.5pt]
- stick to crude measurement methods: `time(1)`/`clock(3)`
- use a number of elements suitable for this measurement style

Intermezzo 1: Assignment 1

- please write a `Makefile` if you didn't (not `CMakeLists.txt`)
- please ensure optimisation are enabled (at least `-O2`)
- `make hw1` should create a binary called `hw1`
  - this also means you can't use `hw1` as a directory name
  - try `hw1.src` or such instead
- `$(CC)` is the C compiler, not C++
  - use `$(CXX)` or just `c++`
  - same for `CFLAGS` vs `CXXFLAGS`
- if you want C++17, please use `-std=c++1z`
  - `-std=c++17` does not work on clang 4.0 or older
- none of the above will incur the -1pt penalty for being late
  - but please do fix those issues ASAP
  - next time these issues will no longer get an exception

Part 3: Micro-Benchmarking and Statistics

Motivation

- there's a gap between high-level code and actual execution
- the gap has widened over time
  - higher-level languages & more abstraction
  - more powerful optimisation procedures
  - more complex machinery inside the CPU
  - complicated cache effects
- it is very hard to predict actual performance

Challenges

- performance is very deterministic in theory
- this is not the case in practice
  - time-sharing operating systems
  - cache content and/or swapping
  - power management, CPU frequency scaling
  - virtual machines
- both micro (unit) and system benchmarks are affected

Unit vs System Benchmarking

- a benchmark only gives you one number
- it is hard to find causes of poor performance
- unit benchmarks are like unit tests
  - easier to tie causes to effects
  - faster to run (minutes or hours vs hours or days)
  - easier to make parametric

Isolation vs Statistics

- there are many sources of measurement errors
- some are systematic, others are random (noise)
- noise is best fought with statistics
- but statistics can’t fix systematic errors
- benchmark data is not normally distributed

Bootstrap

- usual statistical tools are distribution-dependent
- benchmark data is distributed rather oddly
- idea: take many random re-samplings of the data
- take the 5th and 95th percentiles as the confidence interval
- this is a very robust (if stochastic) approach

Using `brick-benchmark`

- implements `fork`-based benchmark isolation
- uses bootstrapping to correctly quantify noise
- uses `clock_gettime` to get precise timings
- adaptive – if the CI is reasonably tight, stop iterating
- simple registration API
- example use – look for `SelfTest` in `brick-benchmark`

Exercise 2

- compare 3 stack implementations
  - `std::vector`, `std::deque` and `std::list`
  - parametrise the benchmark by number of items inserted
  - and by the maximum size of the stack
  - randomise whether to remove or insert
  - insert needs to have a higher probability
- same for queues, but only `std::deque` and `std::list`
- (optional) implement radix sort for integral types
  - compare with `std::sort` on different sequence sizes

Homework 2

- implement `erase()` for both your hash tables [1pt]
- write micro-benchmarks for your hash tables [1pt]
  - also include `std::set` and `std::unordered_set`
  - plot time-per-insert vs number of inserts for each
  - come up with a benchmark for `erase`
- implement a generator of random graphs (of a given size)
- implement BFS using the best hash table and best queue [1pt]

Part 4: The Memory Hierarchy

- many levels of ever bigger, ever slower memories
- CPU registers: very few, very fast (no latency)
- L1 cache: small (100s of KiB), plenty fast (~4 cycles)
- L2 cache: still small, medium fast (~12 cycles)
- L3 cache: ~2-32 MiB, slow-ish (~36 cycles)
- L4 cache: (only some CPUs) ~100 MiB (~90 cycles)
- DRAM: many gigabytes, pretty slow (~200 cycles)

- NVMe: ~10k cycles
- SSD: ~20k cycles
- spinning rust: ~30M cycles
- RTT to US: ~450M cycles

Paging vs Caches

- page tables live in slow RAM
- address translations are very frequent
- and extremely timing-sensitive
- TLB small, very fast address translation cache

- process switch TLB flush
- but: Tagged TLB, software-managed TLB
- typical size: 12 - 4k entries
- miss penalties up to 100 cycles

Additional Effects

- some caches are shared, some are core-private
- out of order execution to avoid waits
- automatic or manual (compiler-assisted) prefetch
- speculative memory access
- ties in with branch prediction

Some Tips

- use compact data structures (vector > list)
- think about locality of reference
- think about the size of your working set
- code size, not just speed, also matters

See Also

- `cpumemory.pdf` in study materials
  - somewhat advanced and somewhat long
  - also very useful (the title is not wrong)
  - don’t forget to add 10 years
  - oprofile is now `perf`
- `http://www.7-cpu.com` CPU latency data

Exercise 3

- write benchmarks that measure cache effects

Some Ideas

- walk a random section of a long `std::list`
- measure time per item in relation to list size
- same but with a `std::vector`
- same but access randomly chosen elements (vector only)

Some Issues

- `uniform_int_distribution` has odd timing behaviour
- but we don't really care about uniformity

- you may need to fight the optimiser a bit
- especially make sure to avoid undefined behaviour

- indexing vs iteration have wildly different behaviour
- shuffling your code slightly can affect the results a lot

Part 5: Profiling I, `callgrind`

Why profiling?

- it's not always obvious what is the bottleneck
- benchmarks don't work so well with complex systems
- performance is not quite composable
- the equivalent of `printf` debugging isn't too nice

Workflow

1. use a profiler to identify expensive code
   - the more time program spent doing X,
   - the more sense it makes to optimise X
2. improve the affected section of code
   - re-run the profiler, compare the two profiles
   - if satisfied with the improvement, goto 1
   - else goto 2

What to Optimise

- imagine the program spends 50 % time doing X
  - optimise X to run in half the time
  - the overall runtime is reduced by 25 %
  - good return on investment
- law of diminishing returns
  - now only 33 % of time is spent on X
  - cutting X in half again only gives 17 % of total
  - and so on, until it makes no sense to optimise X

Flat vs Structured Profiles

- flat profiles are easier to obtain
- but also harder to use
  - just a list of functions and cost
  - the context & structure is missing
- call stack data is a lot harder to obtain
  - endows the profile with very rich structure
  - reflects the actual control flow

`cachegrind`

- part of the `valgrind` tool suite
- dynamic translation and instrumentation
- based on simulating CPU timings
  - instruction fetch and decode
  - somewhat abstract cost model
- can optionally simulate caches
- originally only flat profiles

`callgrind`

- records entire call stacks
- can reconstruct call graphs
- very useful for analysis of complex programs

`kcachegrind`

- graphical browser for callgrind data
- demo

Exercise 4

- there's a simple BFS implementation in study materials
- you can also use/compare your own BFS implementation
- don't forget to use `-O2 -g` or such when compiling
- generate a profile with `cachegrind`
- load it up into `kcachegrind`
- generate another, using `callgrind` this time & compare

Exercise 4 (cont'd)

- add cache simulation options &c.
- explore the knobs in `kcachegrind`
- experiment with the size of the generated graph
- optimise the BFS implementation based on profile data

Homework 3

- implement a real-valued matrix data structure [1pt]
- implement 2 matrix multiplication algorithms [1pt]
  - natural order
  - cache-efficient order
- compare the implementations using benchmarks [1pt]
- the output should be again `gnuplot` sources on stdout

Part 6: Tuning for the Compiler

Goals

- write high-level code
- with good performance

What We Need to Know

- which costs are easily eliminated by the compiler?
- how to make best use of the optimiser (with minimal cost)?

How Compilers Work

- read and process the source text
- generate low-level intermediate representation
- run IR-level optimisation passes
- generate native code for a given target

Intermediate Representation

- for C++ compilers typically a (partial) SSA
- reflects CPU design / instruction sets
- symbolic addresses (like assembly)
- explicit control and data flow

IR-Level Optimiser

- common sub-expression elimination
- loop-invariant code motion
- loop strength reduction
- loop unswitching
- sparse conditional constant propagation
- (regular) constant propagation
- dead code elimination

Common Sub-expression Elimination

- identify redundant (& side-effect free) computation
- compute the result only once & re-use the value
- not as powerful as equational reasoning

Loop-Invariant Code Motion

- identify code that is independent of the loop variable
- and also free of side effects
- hoist the code out of the loop
- basically a loop-enabled variant of CSE

The Cost of Calls

- prevents CSE (due to possible side effects)
- prevents all kinds of constant propagation

Inlining

- removes the cost of calls
- improves all intra-procedural analyses
- inflates code size
- only possible if the IR-level definition is available

See also: link-time optimisation

The Cost of Abstraction: Encapsulation

- API or ABI level?
- API: cost quickly eliminated by the inliner
- ABI: not even LTO can fix this
- ABI-compatible setter is a call instead of a single store

The Cost of Abstraction: Late Dispatch

- used for `virtual` methods in C++
- indirect calls (through a vtable)
- also applies to C-based approaches (`gobject`)
- prevents (naive) inlining
- compilers (try to) devirtualise calls

Exercise 5

- start with `bfs.cpp` from study materials
- make a version where `edges()` is in a separate C++ file
- you will need to use `std::function`
- try a compromise using a visitor pattern
- compare all three approaches using benchmarks

Intermezzo 2: Competition & Homework

Competition

- download `competition1.tar.gz` from study materials
- run `make personalize I=xx` where `xx` is your initials
- in `xx.hpp`, implement a set of `char`
  - must support `insert`, `erase` and `count`
  - operator `&` (for intersection of two sets)
  - operator `|` (for union of two sets)
- `make check` to run unit tests
- `make bench` to run benchmarks

Homework 4

- implement a set of `uint16_t` using a bitvector [1pt]
  - with insert, erase, union and intersection
- the same using a nibble-trie [1pt]
  - a trie with out-degree 16 (4 bits)
  - should have a maximum depth of 4
  - implement insert and union
- compare the two implementations [1pt]

Part 7: Understanding the CPU

The Simplest CPU

- in-order, one instruction per cycle
- sources of inefficiency
  - most circuitry is idle most of the time
  - not very good use of silicon
- but it is reasonably simple

Design Motivation

- silicon (die) area is expensive
- switching speed is limited
- heat dissipation is limited
- transistors cannot be arbitrarily shrunk
- “wires” are not free either

The Classic RISC Pipeline

- fetch – get instruction from memory
- decode – figure out what to do
- execute – do the thing
- memory – read/write to memory
- write back – store results in the register file

Instruction Fetch

- pull the instruction from cache, into the CPU
- the address of the instruction is stored in PC
- traditionally does branch "prediction"
  - in simple RISC CPUs always predicts not taken
  - this is typically not a very good prediction
  - loops usually favour taken heavily

Instruction Decode

- not much actual decoding in RISC ISAs
- but it does register reads
- and also branch resolution
  - might need a big comparator circuit
  - depending on ISA (what conditional branches exist)
  - updates the PC

Execute

- this is basically the ALU
  - ALU = arithmetic and logic unit
- computes bitwise and shift/rotate operations
- integer addition and subtraction
- integer multiplication and division (multi-cycle)

Memory

- dedicated memory instructions in RISC
  - load and store
  - pass through execute without effect
- can take a few cycles
- moves values between memory and registers

Write Back

- write data back into registers
- so that later instructions can use the results

Pipeline Problems

- data hazards (result required before written)
- control hazards (branch misprediction)
- different approaches possible
  - pipeline stalls (bubbles)
  - delayed branching
- structural hazards
  - multiple instructions try to use a single block
  - only relevant on more complex architectures

Superscalar Architectures

- more parallelism than a scalar pipeline
- can retire more than one instruction per cycle
- extracted from sequential instruction stream
- dynamically established data dependencies
- some units are replicated (e.g. 2 ALUs)

Out-of-order execution

- tries to fill in pipeline stalls/bubbles
- same principle as super-scalar execution
  - extracts dependencies during execution
  - execute if all data ready
  - even if not next in the program

Speculative Execution

- sometimes it's not yet clear what comes next
- let's decode, compute etc. something anyway
- fills in more bubbles in the pipeline
- but not always with actual useful work
- depends on the performance of branch prediction

Take-Away

- the CPU is very good at utilising circuitry
- it is somewhat hard to write “locally” inefficient code
- you should probably concentrate on non-local effects
  – non-local with respect to instruction stream
  – like locality of reference
  – and organisation of data in memory in general
  – also higher-level algorithm structure

Exercise 6

- implement a brainfuck interpreter
- try to make it as fast as possible
- see wikipedia for some example programs

Bonus Homework

- write a brainfuck amd64 compiler
- 2 points for emitting symbolic assembly
- 1 extra for emitting binary code

Part 8: Exploiting Parallelism

Hardware vs Software

- hardware is naturally parallel
- software is naturally sequential
- something has to give
  - depends on the throughput you need
  - eventually, your software needs to go parallel

Algorithms

- some algorithms are inherently sequential
  - typically for P-complete problems
  - for instance DFS post-order
- which algorithm do you really need though?
  - topological sort is much easier than post-order
- some tasks are trivially concurrent
  - think map-reduce

Task Granularity

- how big are the tasks you can run in parallel?
  - big tasks = little task-switching overhead
  - small tasks = easier to balance out
- how much data do they need to share?
  - shared memory vs message passing

Distributed Memory

- comparatively big sub-tasks
- not much data structure sharing (small results)
- scales extremely well (millions of cores)

Shared Memory

- small, tightly intertwined tasks
- sharing a lot of data
- scales quite poorly (hundreds of cores)

Caches vs Parallelism

- different CPUs are connected to different caches
- caches are normally transparent to the program
- what if multiple CPUs hold the same value in cache
  - they could see different versions at the same time
  - need cache coherence protocols

Cache Coherence

- many different protocols exist
- a common one is MESI (4 cache line states)
  - modified, exclusive, shared, invalid
  - snoops on the bus to keep up to date
- cheap until two cores hit the same cache line
  - required for communication
  - also happens accidentally

Locality of Reference

- comes with a twist in shared memory
- compact data is still good, but
  - different cores may use different pieces of data
  - if they are too close, this becomes costly
  - also known as false sharing

Distribution of Work

- want to communicate as little as possible
- also want to distribute work evenly
- randomised, spread-out data often works well
  - think hash tables
- structures with a single active point are bad
  - think stacks, queues, counters &c.

Shared-Memory Parallelism in C++

- `std::thread` – create threads
- `std::future` – delayed (concurrent) values
- `std::atomic` – atomic (thread-safe) values
- `std::mutex` and `std::lock_guard`

Exercise 7

- implement shared-memory map-reduce in C++
- make the number of threads a runtime parameter
- check how this scales (wall time vs number of cores)
- use this for summing up a (big) array of numbers
- can you improve on this by hand-rolling the summing loop?

Homework 5

- implement parallel matrix multiplication [2pt]
- compare to your sequential versions [1pt]
  - try with 2 and 4 threads in your benchmarks