C++ Performance

Liam Keegan, SSC

Course Outline

- Performance
  - How algorithms, data structures and hardware affect performance
- Strategy
  - What to optimize, how to do it, and how far to go
- Best practices
  - Broadly applicable c++ strategies and idioms for performance
- Other tricks
  - Other c++ tricks, only applicable in certain situations, use with care
- Code examples
  - Optimize some code samples for performance

Performance

Motivation

- We use c++ because we need performance
  - Otherwise we could (should?) use a higher-level language, e.g. Python
- Writing fast code is simple
  - Need to use good ***algorithms*** for ***efficiency*** (do less work)
  - Need to use good ***data structures*** for ***speed*** (do work faster)
- Writing fast code is difficult
  - Often these two needs are in conflict with each other
  - So the answer to “what is the fastest way to do this?” is “it depends”
- Benchmarking can help
  - With the question “which method is fastest?”
  - With the question “where and why is my code slow?”

Algorithms

- Complexity
  - Count operations required to process N items
    - E.g. 2 N + 5 N log(N)
  - For large N, term with highest power of N dominates
  - “Big-O” algorithmic complexity is the highest power of N
    - E.g. O(N log(N))
- Memory
  - Required memory is also counted and expressed in Big-O notation
- Optimal algorithms
  - For many problems, can prove an algorithm is “optimal”
  - This means is has the same Big-O cost as the best possible algorithm

Algorithm goal: reduce complexity

- Generally we want the most efficient algorithm
  - I.e. the one with the smallest Big-O complexity
- Often there a multiple “optimal” algorithms
  - Different pros and cons depending on your data and your hardware
- Although sometimes a less efficient algorithm can be better
  - if memory is a constraint, and there is a less efficient algorithm that requires less memory
  - if the big-O *coefficient* is much smaller
    - E.g. unless N is very large, 1e9 * log(N) is larger than N
  - if it allows for a much faster implementation on your hardware
    - E.g. more fast operations can be better than a few slow operations
  - if it allows for parallelization over threads / cores / nodes
  - etc

Hardware

- CPU gets some data, operates on it:
- Get the data from
  - L1 cache (32kb): 1 ns
  - L2 cache (256kb): 3 ns
  - L3 cache (32mb): 20 ns
  - RAM (32gb): 100 ns
  - SSD (1tb): 10000 ns
- Do the operation
  - CPU core cycle: 1 ns
  - SIMD: apply same instruction to multiple data in the same operation
  - Multi-core: a CPU typically has several cores
- CPU spends a lot of time idle, waiting for data

Cache

- Cache misses are very expensive
  - L1 cache: 1 ns wait + 1 ns work
  - RAM: 100 ns wait + 1 ns work
- Cache lines
  - Cache data is organised in lines of 64 bytes
  - CPU gets the whole cache line when it asks for an item of data
- SIMD
  - A single CPU instruction can operate on multiple items of data
  - Eg AVX512 instructions can operate on all 64 bytes in a single operation
- Pre-fetching
  - CPU will also try to guess what data will be needed next
  - If you are iterating linearly over contiguous data, will nearly always predict correctly

Hardware goal: avoid cache misses

- Spatial locality of data
  - Store data contiguously, and in the order in which it will be accessed
  - Also known as “Data-oriented design”
    - or old-fashioned C-style arrays of data
    - or SoA (struct-of-arrays) vs AoS (arrays-of-struct)
  - Why? So that it is more likely to already be in the cache
- Temporal locality of algorithms
  - When accessing the same data again, do it sooner rather than later
  - Why? So that it is more likely to still be in the cache

General Performance strategy

- For large problems
  - We care about the efficiency of our ***algorithms***
  - We want to do less work
- For small problems
  - We care about cache friendly ***data structures***
  - We want to do our work fast
- Sometimes a large problem can also be many small problems
  - Use an ***efficient*** algorithm to reduce a large problem to many small problems
  - Use a ***fast*** cache-friendly method to solve each small problem
- Algorithms and data structures are interconnected
  - But helpful to consider their impact separately
  - Helps to understand the tradeoffs you are making with your choices

Strategy

Strategy

- Write code with performance in mind
  - Apply best practices, use good data structures & algorithms
  - But don’t prematurely optimize or sacrifice readability or maintainability (or correctness!)
- Then identify what needs to be improved
  - Profile your code, identify hotspots
  - You need benchmarks to understand performance impact of changes
- Refactor to improve performance
  - Ideally by using better data structures / algorithms / libraries
  - Sometimes by replacing a library with custom micro-optimized code
    - Consider development effort vs benefit
    - Consider maintenance: data / hardware / compiler changes
    - Consider reliability: library typically more robust and better tested

Best Practices

Use appropriate data structures

- If in doubt, use a std::vector
    - Contiguous data storage: cache friendly data structure
    - Should be your default choice
- If you have key/value pairs, use a map
    - std::map is ok but not cache-friendly
    - std::unordered_map is typically faster
    - Libraries like abseil, folly etc offer faster hash maps
    - Or, if you are mostly reading from a map
        - consider replacing it with a sorted vector
- Avoid std::list

Use appropriate algorithms

-

Avoid unnecessary copies

- Don’t pass (large) objects by value, pass by reference or pointer
- Do return objects by value: RVO (guaranteed with c++17)
    - Declare object to be returned at start of function
    - Return it by value at end of function
    - Compiler will construct it in-place at function call site (RVO)
    - Note: older “optimization” was to move-return the object
        - This was a performance optimization for older compilers
        - Now this is actually a pessimization
            - forces compiler to do a move instead of constructing in-place

Other Tricks

Other tricks

- Disable exceptions
- Enable fast math
- Use likely/unlikely

Code Samples

Code samples

```
git clone --recursive
https://github.com/ssciwr/high-performance-cpp.git
cd high-performance-cpp
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
ctest
./bench/bench
```

Summary

Summary

- Good performance comes from algorithms and data structures
- Write (correct, clear, maintainable) code with performance in mind
- Have a strategy to identify where and what to improve
- Always benchmark changes
- Always keep in mind tradeoffs and costs vs benefits

Recommended resources

Performance

- Mike Acton talk on data oriented design
  - Data-Oriented Design and C++
- Ulrich Drepper paper on memory and cache
  - What Every Programmer Should Know About Memory
  - Not for “every programmer” but excellent info on RAM / cache / performance
  - From 2007 but still very relevant
- A curated list of c++ performance-related resources
  - AwesomePerfCpp

Recommended resources

Benchmarking

- Chandler Carruth talks on benchmarking/performance
  - Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!
  - Efficiency with Algorithms, Performance with Data Structures
  - Going Nowhere Faster
- Fedor Pikus talk on branchless programming
  - Branchless Programming in C++

Profiling

- http://pramodkumbhar.com/2017/04/summary-of-profiling-tools/

More recommended resources

Hash maps

- Malte Skarupke talk on hash map implementations and performance
  - You Can Do Better than std::unordered_map
- Matt Kulukundis talks on Google’s hash map
  - Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step
  - Abseil's Open Source Hashtables: 2 Years In