C++ Benchmarking

Liam Keegan, SSC

Course Outline

- Performance
    - How to get good performance
    - Why benchmarking is important
- Micro-benchmarks
    - Run a tiny snippet of code on real hardware
    - Do it many times & see how long it takes
- Profiling
    - Run the entire code on real hardware, occasionally check what it's doing
    - Non-deterministic statistical sampling of events, but on real hardware
- Simulation
    - Run entire code on a virtual machine & measure every single operation
    - Deterministic measurement of all events, but not real hardware

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

Map implementation performance

- Binary Tree: O(log(N)) to find an item
  - Good algorithmic efficiency
  - Terrible cache locality
  - Use case: large N
- Unsorted Vector: O(N) linear traversal to find an item
  - Terrible algorithmic efficiency
  - Good cache locality
  - Use case: small N
- Hash table: O(1)
  - Excellent algorithmic efficiency
    - Although not guaranteed - need a good hash function
  - Cache locality depends on implementation
  - Use case: large N

Micro-benchmarks

Micro-benchmarks

- Run a tiny snippet of code on real hardware
- Do it many times & see how long it takes
- Good to get accurate results for specific functions
- May not be so close to “real world” scenarios
- Have to take care compiler doesn’t optimize away the supposed work!
- Not deterministic: but can repeat many times & measure statistical significance
- Different possible metrics
  - Average time: most common and most relevant for many use cases
  - Median time: less noisy than the average, not affected by outliers
  - Fastest time: e.g. if investigating peak performance
  - Slowest time: e.g. if latency / worst-case scenario is important

Micro-benchmarking tools

- Google benchmark
  - Widely used and actively maintained
  - See also Quick-bench (online version a bit like compiler explorer)
- Celero
  - Actively maintained
- Nonius
  - Older, less/not actively maintained
- Catch2
  - Some integration of nonius to add benchmarks to unit tests
- Do it yourself solution
  - E.g. using `std::chrono::high_resolution_clock`
  - Not recommended

Google benchmark minimal example

```
#include <benchmark/benchmark.h>

static void bench_map(benchmark::State &state) {
    Map map; // initialize data
    for (auto _ : state) {
        map.find(keys[0]); // code to benchmark
    }
}

BENCHMARK(bench_map);

BENCHMARK_MAIN();
```

Live coding time

```
git clone --recursive https://github.com/ssciwr/cpp-benchmarking.git
cd cpp-benchmarking
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
ctest
./bench/bench1
```

Benchmark 1

```
for (auto _ : state) {
    map.find(keys[0]);
}
```

```
bench_small_map/1          0.000 ns        0.000 ns   1000000000
bench_small_map/2          0.000 ns        0.000 ns   1000000000
bench_small_map/4          0.000 ns        0.000 ns   1000000000
```

- Looks bogus: Zero run-time, no N-dependence
- Why? Compiler optimizes away the entire statement

Benchmark 2

```
for (auto _ : state) {
    benchmark::DoNotOptimize(map.find(keys[0]));
}
```

```
bench_small_map/1          0.990 ns        0.990 ns     691986485
bench_small_map/2           1.00 ns         1.00 ns     705385115
bench_small_map/4          0.989 ns        0.989 ns     703257489
```

- Force compiler to put result into register or memory
- Non-zero runtime, but still no N-dependence
- Why? Always looking for the first key!

Benchmark 3

```
for (auto _ : state) {
    benchmark::DoNotOptimize(map.find(keys[n/2 + 1]));
}
```

```
bench_small_map/1          1.14 ns          1.14 ns    600298793
bench_small_map/2          1.38 ns          1.38 ns    507900144
bench_small_map/4          2.51 ns          2.51 ns    278833095
```

- Force compiler to put result into register or memory
- Ask for a key roughly in the middle of the vector
- Get reasonable-looking N-dependence

Benchmark 4

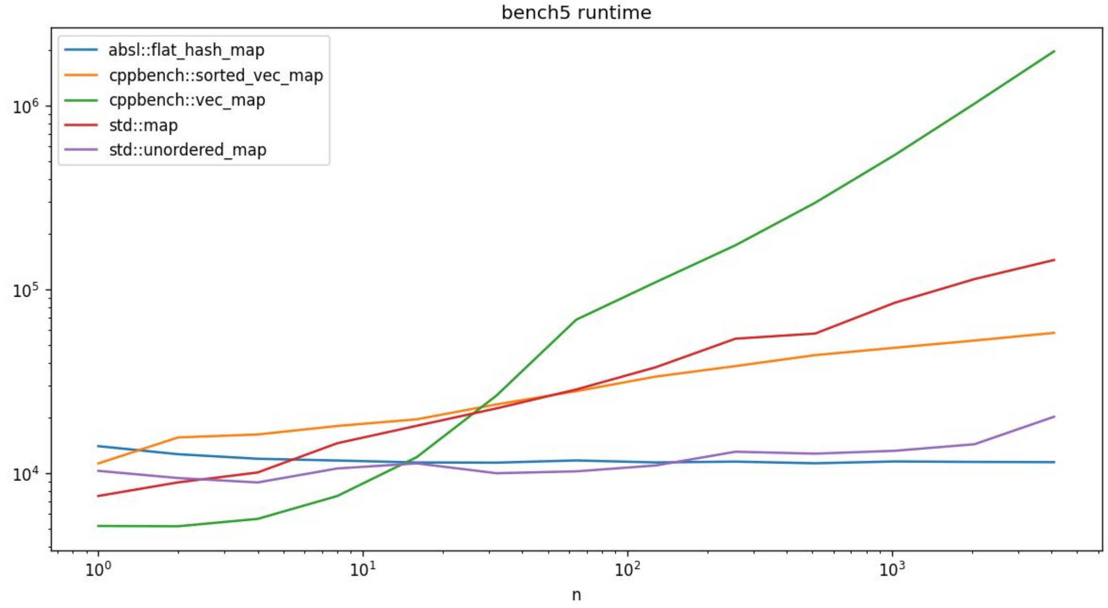
```
for (auto _ : state){
    for (auto key : keys) {
        benchmark::DoNotOptimize(map.find(key));
    }
}
```

```
bench_small_map/1            1.34 ns         1.34 ns    495155683
bench_small_map/2            3.21 ns         3.21 ns    218922642
bench_small_map/4            6.53 ns         6.53 ns    105903854
```

- Force compiler to put result into register or memory
- Loop over all keys: now doing N finds per benchmark iteration
- Get average performance over all inputs

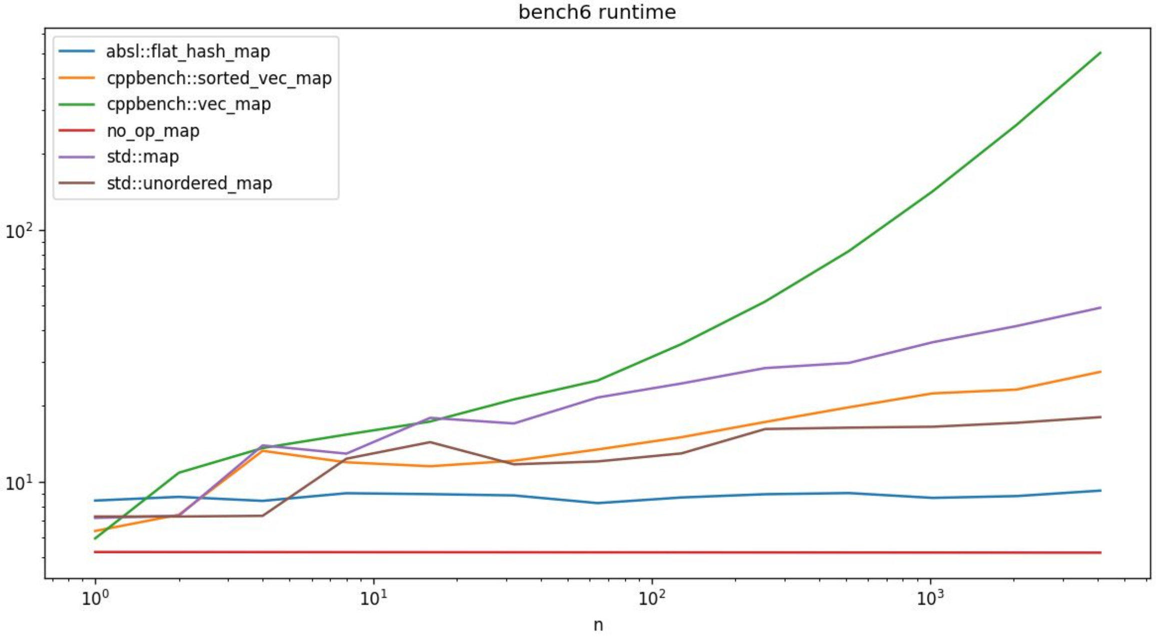
Benchmark 5

- Compare with std::map, std::unordered_map implementations
- Do the same number of find operations per benchmark iteration



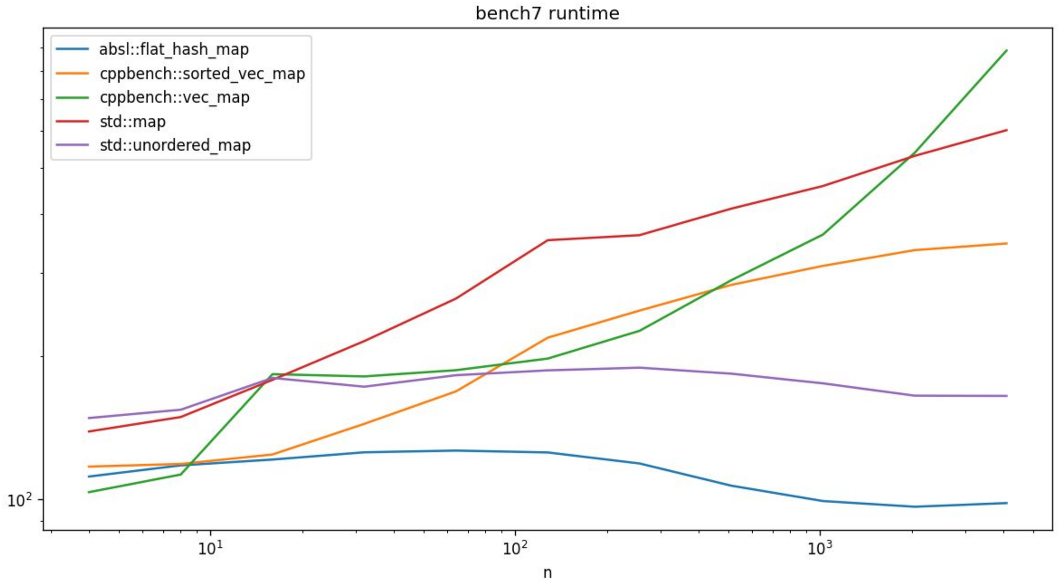
Benchmark 6

- Pick a random key at each iteration (more realistic, but adds overhead)
- Add a “no-op” Map to measure this overhead



Benchmark 7

- Try to simulate a “cold cache scenario”
  - Generate millions of identical maps (a few gig of RAM)
  - Pick a random map (probably not already in cache) for each benchmark iteration



Micro-benchmarking tips

- Compile in release mode
- Reduce the noise
    - Disable CPU turbo-boost / frequency scaling
    - Avoid running anything else at the same time
    - Repeat benchmarks to estimate noise, statistical significance of differences
- Try to be sure you’re measuring what you intend to
    - Check the inputs and outputs are correct
    - Check the compiler is not optimizing away the work
    - Try to use or generate representative realistic input data
    - Compare timings with a no-op version of your operation / function
    - Look at the scaling as you vary the size of the input data
    - Try varying the order of the input data
    - Cold cache or hot cache scenario

Profiling

Profiling

- Run real production code on real hardware
- Occasionally check what it’s doing
    - Not too often to avoid interfering with execution / slowing it down!
- Good to get an overview of real-world bottlenecks
- Not deterministic
- Only sample a tiny fraction of events
    - But likely to find hotspots this way

Profiling Tools

- Perf
  - No special instrumentation or re-compilation of code required
  - Relatively simple to use, pre-installed on most linux distributions
  - But linux only, and need sudo permissions:
    https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html
- Gperftools
- Intel VTune
- Likwid
- Tracy
- and many more...

Scientific Software Center

Compiling for profiling

- Generally want to compile in Release mode
  - Goal is to run your code exactly as in real-world setting
- Additional flags to improve profile output at minimal cost:
  - -g : include debugging symbols (e.g. function names)
    - CMAKE_BUILD_TYPE = RelWithDebInfo
  - -fno-omit-frame-pointer : don't omit frame pointers (i.e. better stack traces)
- Any additional requirements for your profiler
  - Link to their library?
  - Include their header?
  - Annotate your code?
  - Set environment variables?

Perf stat

- perf stat ./app
  - default performance counters
- perf list
  - show all available counters
- perf stat -e L1-dcache-load-misses,L1-dcache-load ./app
  - how often do we have a L1 data cache miss
- perf stat -e branches,branch-misses ./app
  - how often do we have a branch prediction miss

Perf stat example

```
perf stat -e L1-dcache-load-misses,L1-dcache-load \
./bench/bench6 \
--benchmark_filter=".*:vec_map.*/16\$" \
--benchmark_min_time=2
```

- First line: Measure cache misses using perf stat
- Others: run n=16 vec_map benchmark from bench6 for at least 2 secs

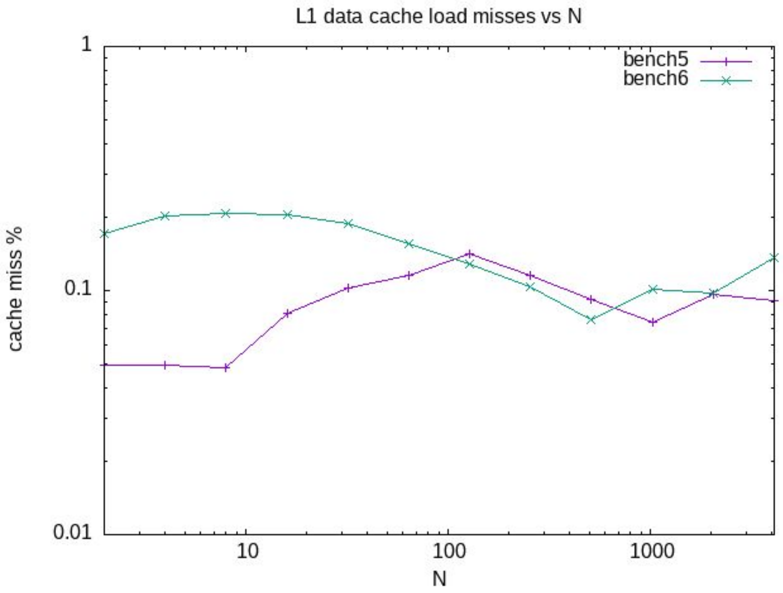
```
627,014       L1-dcache-load-misses     #    0.01% of all L1-dcache accesses
8,377,940,732       L1-dcache-load
```

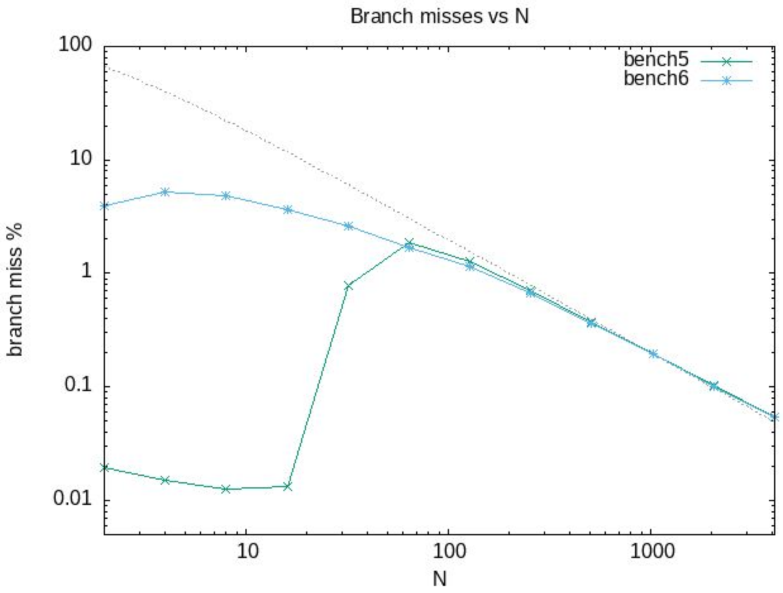
Branch prediction

- CPU constructs a pipeline of instructions
  - Often “out of order” i.e. not evaluated in the order you wrote them
  - E.g. part of iteration n+1 in a loop may be calculated in iteration n
  - You need to know what instructions are coming next to be able to do this
- At every branch (e.g. if statement) in the code
  - CPU doesn’t know where execution will jump to next until condition is evaluated
  - This stalls the pipeline
- So it tries to guess the most likely outcome (branch prediction)
  - Constructs the pipeline based on guessing where the branch will jump to
- Sometimes it guesses wrong
  - This is called a “branch miss” (similar to a cache miss)
  - CPU has to throw away / undo a lot of potentially incorrect operations
  - Then find and execute the correct instructions from the correct branch
  - This is expensive (also similar to a cache miss)

Perf branch prediction

- Branch predictor is outsmarting our benchmark in bench5





Perf record

- perf record ./app && perf report
  - see annotated assembly with % of time for each line
    - h: show shortcuts
    - H: go to hotspot
    - Enter: go to function / follow jump

```
51.47 │1d0:   cmp    %ecx,(%rbx,%rsi,4)
 0.16 │     ↑ je     1a0
 2.94 │       inc    %rsi
 0.01 │       cmp    %rsi,%rax
41.29 │     ↑ jne    1d0
      │     ↑ jmp    1a7
```

```
./bench/bench5 --benchmark_filter=".*:vec_map.*/1024\$" --benchmark_min_time=2
```

Perf visualization

- Raw assembly can be overwhelming
- Can be easier to interpret a call graph, flame graph, etc
- Options (all inconvenient in different ways)
  - https://github.com/KDAB/hotspot
    - Works directly on perf.data, but output seems incomplete / wrong
  - https://github.com/brendangregg/FlameGraph
    - Use supplied perl script to convert trace data first
  - https://github.com/google/pprof
    - But also need https://github.com/google/perf_data_converter
  - https://www.eclipse.org/tracecompass
    - But you need to re-compile perf with CTF support

Profiling tips

- Reduce the noise
  - Disable CPU turbo-boost / frequency scaling
  - Avoid running anything else at the same time
- First step: look at cache misses
  - If there are a lot of these, nothing else matters much
- Look at branch misses
  - Sometimes Profile Guided Optimization (PGO) can help
  - Very rarely you can improve them by hand (__builtin_expect, [[likely]], etc)
  - But hard to do better than the CPU (and easy to make things worse!)
- Look at hotspots
  - Lines of code that are called a lot and/or slow
- Profile your benchmarks
  - Can help understand if they are testing what you want them to

Simulation

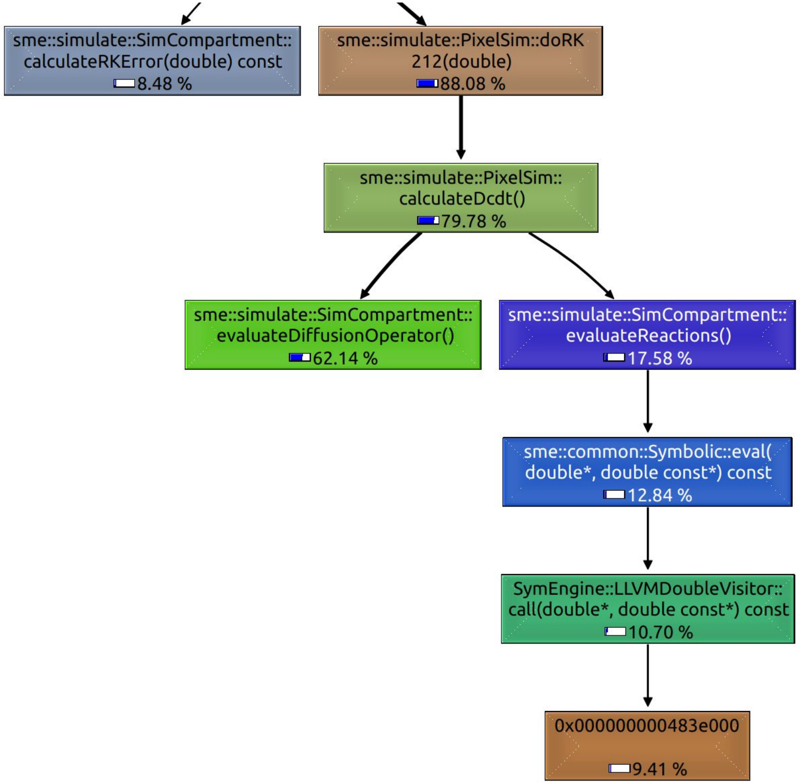
Simulation

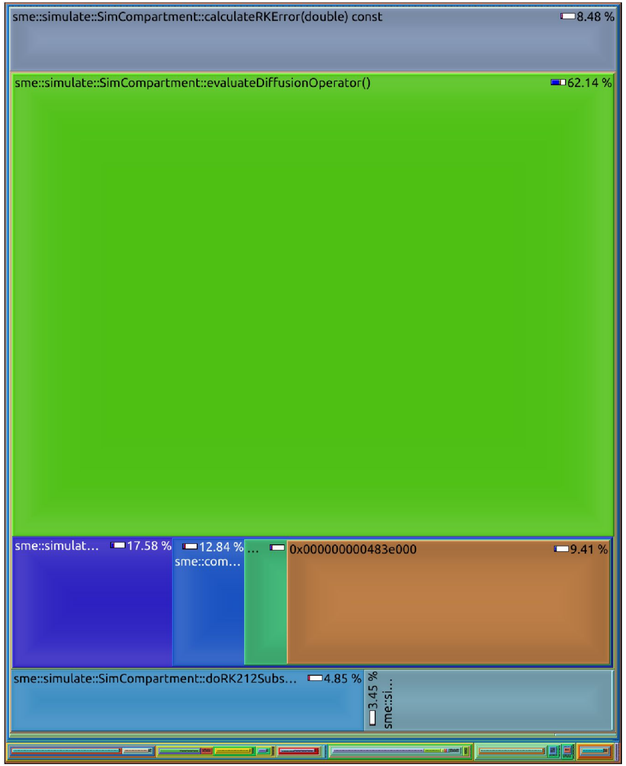
- Run real production code on a virtual machine
  - Simulated CPU, simulated cache
- Record every cpu operation, every cache read, etc
  - Doesn’t affect (simulated) speed of operations
- Deterministic
  - No statistical variation, not affected by other running processes
- But: not real hardware
  - No guarantee performance implications are the same
- Also: typically vastly slower than real-time (e.g. 50x slower)
  - Can be inconvenient / impractical for certain applications

Callgrind/cachegrind

- Part of valgrind set of tools
  - Default tool is memcheck, but there are others!
- valgrind –tool=cachegrind ./app
  - Command line output of simulated cache hits/misses
- valgrind –tool=callgrind --simulate-cache=yes ./app
  - Generates call-graph data and cache hits/misses
- kcachegrind
  - GUI to visualize above data

Example call graph (cycles)





Perf report for same code

```
61.55%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::evaluateDiffusionOperator
 8.83%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::calculateRKError
 4.98%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::evaluateReactions
 4.36%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::doRK212Substep2
 3.51%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::doRK212Substep1
 2.11%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69035
 1.59%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69004
 1.52%  spatial-cli  spatial-cli         [.] SymEngine::LLVMDoubleVisitor::call
 1.37%  spatial-cli  spatial-cli         [.] sme::common::Symbolic::eval
 1.00%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef6904b
 0.78%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef6901b
 0.55%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef6900c
 0.52%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69031
 0.44%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69047
 0.41%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef6901f
 0.29%  spatial-cli  libc.so.6           [.] __GI_____strtod_l_internal
 0.29%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69039
 0.26%  spatial-cli  [JIT] tid 263317    [.] 0x00007fe92ef69023
 0.22%  spatial-cli  spatial-cli         [.] sme::simulate::SimCompartment::undoRKStep
```

Scientific Software Center

Simulation tips

- Ignore the noise!
  - CPU turbo-boost, other running processes not relevant
- Otherwise, same tips as profiling
  - First step: look at cache misses
    - If there are a lot of these, nothing else matters much
  - Look at branch misses
    - Sometimes Profile Guided Optimization (PGO) can help
    - Very rarely you can improve them by hand (__builtin_expect, [[likely]], etc)
    - But hard to do better than the CPU (and easy to make things worse!)
  - Look at hotspots
    - Lines of code that are called a lot and/or slow
  - Profile your benchmarks
    - Can help understand if they are testing what you want them to
- Remember it’s a simulation - compare results with actual profiling data

Summary

Summary

- Benchmarking is vital but also difficult to do reliably
- Important not to blindly take one metric or benchmark as the “truth”
- We covered three complementary approaches
    - Micro-benchmarking
    - Profiling
    - Simulation
- Each have their own strengths and weaknesses
- Together they allow you to **understand** the performance of your code
- Also where and how to **improve** the performance of your code

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++

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

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