Profiling your code

Raul P. Pelaez

March 19, 2026

Contents

1 Introduction

In some circumstances we will find ourselves needing to make our code more efficient. Perhaps we are trying to scale our application up (to handle more users, or more data), or maybe a graphical application is becoming unresponsive or we need our results faster. In these cases, we need to profile our code to identify bottlenecks and optimize them. This lesson will guide you through the basics of profiling C++ applications and introduce some generic optimization techniques such as function inlining, cache optimization, loop unrolling and vectorization.

Advice

Premature optimization is the root of all evil
Back in the 1960s, Donal Knuth wrote the following in his book "The Art of Computer Programming".

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

This is still relevant today, one of those ancient pieces of wisdom in software development that we seem to constantly forget and rediscover. The idea is that we should focus on writing clear, maintainable code first, and only optimize when (and where) necessary.
This is the reason why this is one of the last lessons in the course.
While it is a good idea to design our software around the idea of it being performant, we should not spend time optimizing code that is not a bottleneck. If we make smart choices during our implementation the code will be performant enough for most cases. For instance, if you are trying to sort a vector, use `std::sort` instead of writing a naive $O(N^2)$ version yourself.
On the other hand, compilers are insanely good at optimizing code for us, so instead of trying to outsmart the compiler, we should strive to make things as clear as possible for it.

2 Measuring time

C++ comes with the `<chrono>` library which provides a high-resolution clock that can be used to measure time. This is the recommended way to measure time in C++. The chrono library is quite deep and has a lot of functionality, but we are going to use just a couple of functions from it.

Source code

```cpp
#include <iostream>
#include <chrono>
using namespace std::chrono;
int main() {
  auto start = high_resolution_clock::now();
  // Code to measure
  auto end = high_resolution_clock::now();
  auto elapsed = end - start;
  std::cout << "Elapsed time: " << elapsed.count() << " s\n";
  auto elapsed_ms = duration_cast<milliseconds>(elapsed);
}
```

3 Basics of profiling

Info

Profiling is the process of measuring various aspects of a program's execution to identify areas where performance can be improved. It involves collecting data on function call frequencies, execution times, memory usage, and more.

There are a plethora of tools available for each platform to aid in profiling. In my experience, not of them really "cut it". The one that has been most useful to me in the past is nsight-systems, which is part of the NVIDIA toolkit and focused on CUDA applications.

They all have strong shortcomings in one aspect or another (for instance, valgrind will slow your application to a crawl in order to give you information about call stacks and memory operations). Luckily, we can get around using full-fledged profilers in most circumstances. The point is, after all, to identify bottlenecks and optimize them.
It is also in general unwise for us to focus on the particular workings of a specific application, since they come and go and change between platforms. Instead, we will be focusing on general techniques and wisdoms that can be applied to any application.

Advice

Common Profiling Tools:
For completeness, I list here some popular profiling tools that I encourage you to explore.

- **Linux:**
  Valgrind:
  Callgrind: Profiles function calls and cache performance.
  Massif: Profiles heap memory usage.
  Perf: Performance analysis tool.
- **Windows:**
  Visual Studio Profiler: Integrated profiling in the IDE.
  Intel VTune: Detailed performance analysis, especially on Intel hardware.
- **OSX**:
  Instruments: Performance analysis tool in Xcode.
- **Cross-Platform:**
  gperftools: CPU and heap profiler from Google.

Setting up for profiling:

When we compile our code, we are translating it from human-readable code to machine code. In this process the human-readable C++ code is lost, transformed into a series of assembly instructions specific to the target architecture (the CPU). Note that this assembly code will be different in, say, an ARM processor than in an x86 processor.
When profiling, we want to be able to relate the assembly code back to the original C++ code. To do this, we need to compile our code with "debug symbols". This will make the compiler include information about the original C++ code in the binary, which will be used by the profiler to map the assembly code back to the original C++ code.

Advice

Why not always compile with debug symbols?
In essence, they also make it easier for someone to reverse-engineer your code.

Adding debug symbols amounts to adding the '-g' flag to the compiler. For instance, if we are using g++, we would compile our code like this:

Source code

```
$ g++ -g -o my_program my_program.cpp
```

When investigating performance, we normally wan to compile our code with optimizations turned on. When allowed, the compiler will restructure our code (without changing its

4 Performance considerations

Advice

The compiler knows better than you
Modern compilers are incredibly good at optimizing code. They can do things like function inlining, loop unrolling, and vectorization automatically. When allowed, compilers will reorganized our code in insane ways to make it faster.
Never ever try to outsmart the compiler. Instead, write clear, maintainable code and let the compiler do its job.
In the following, I give you general advice on how to write code that is easier for the compiler to optimize.

Advice

Focus on algorithmic complexity
Your first priority when trying to make code run faster should be to focus on algorithms. For instance, if you are trying to sort a vector, use `std::sort` instead of writing a naive $O(N^2)$ version yourself. Try to identify the places in your logic where $O(N^2)$ or $O(N^3)$ algorithms can be replaced with $O(N \log N)$ or $O(N)$ algorithms.

4.1 Compiler options

The compiler has a series of flags that can be used to optimize the code. The most common ones are:

- '-O3': This flag enables all optimizations. It is the most aggressive optimization level.
- '-O2': This flag enables most optimizations, but not all of them.
- '-march=native': This flag tells the compiler to optimize the code for the specific CPU architecture of the machine where the code is being compiled. Each CPU has a series of instructions that are specific to it, and this flag tells the compiler to use them. Naturally, more specific instructions tend to be more efficient for a given CPU.
- '-ffast-math': This flag tells the compiler to use more aggressive optimizations for floating-point operations. This can lead to faster code, but it can also lead to different results due to floating-point precision issues.
- '-flto': This flag enables link-time optimization. Normally, the compiler optimizes each source file individually. With link-time optimization, the compiler can optimize the whole program at once, which can lead to better optimizations.

4.2 Function inclining

Function inlining replaces a function call with the body of the function itself, eliminating the overhead of the call.
The `inline` keyword **hints** to the compiler that a function should be inlined, but the final decision is up to the compiler.
Example:

Source code

```
inline int add(int a, int b) {
  return a + b;
}
```

Advice

The compiler knows better than you regarding inlining. You do not need to tell the compiler to inline every single time. Use it as a hint to inform the compiler (like when defining a helper function in a library that will not be exposed).
Take this section as an introduction to the concept of inlining, not as an optimization technique.

Advice

Missing functions
Sometimes you will try to debug a program and find that the debugger is not able to step into a function. This is because the function has been inlined by the compiler. The same goes for profiling tools.
When debugging you can simply turn off optimizations (by passing the '-O0' flag to the compiler) to prevent the compiler from inlining functions.
When profiling, one really wants to allow the compiler to optimize the code. One solution is to ask the compiler to not inline specific functions by giving them the `noinline` attribute. Attributes are compiler extensions and not part of the language, but some of them, such as this, are widely supported.
Example:

Source code

```
void __attribute__((noinline)) my_function() {
  // This function will never be inlined
}
```

4.3 Cache

The CPU has a series of caches that are used to store data that is frequently accessed. The caches are much faster than the main memory, but they are also much smaller. This means that if the data we are accessing is not in the cache, we will have to wait for it to be fetched from the main memory, which is much slower.

Advice

Although typically we do not have a way to control the cache directly it is useful to keep in mind that there are usually around three levels of cache between the CPU and the main memory (RAM). The L1 cache is the smallest (a few kilobytes) and fastest, followed by the L2 and L3 caches (a few megabytes).

Info

Data Locality
You should strive to maximize data locality, in both:

- **Spatial Locality:** Accessing data elements stored close to each other.
- **Temporal Locality:** Re-accessing the same data multiple times over a short period.

We can optimize for cache performance by:

- Using contiguous data structures (like arrays/vector) instead of linked lists or maps.
- Accessing memory sequentially to take advantage of prefetching.
- Minimizing jumping around in memory to reduce cache misses.
- Structuring data in a way that benefits access patterns (e.g., Structure of Arrays vs. Array of Structures).

Advice

Structure of Arrays (SoA) vs. Array of Structures (AoS):
When dealing with multiple data elements, you can choose to store them in an array of structures (AoS) or a structure of arrays (SoA). The choice depends on the access patterns of your code.
Array of Structures (AoS):

Source code

```
struct Particle {
  float x, y, z;
  float vx, vy, vz;
};
std::vector<Particle> particles;
```

Structure of Arrays (SoA):

Source code

```
struct Particles {
  std::vector<float> x, y, z;
  std::vector<float> vx, vy, vz;
};
Particles particles;
```

In the AoS layout, all the data for a single particle is stored together. This is good if you are always accessing all the data for a single particle at once. In the SoA layout, all the data for a single property is stored together. This is good if you are always accessing the same property for all particles at once.

4.4 Loop unrolling

Loop unrolling is a technique that involves expanding the loop body to reduce the number of iterations and loop control overhead.
Additionally, loop unrolling can increase instruction-level parallelism and enable further compiler optimizations (like SIMD).
The compiler will consider unrolling loops automatically if it thinks it will be beneficial. However, you can also hint it to do so by using the `unroll` attribute.

Source code

```
constexpr int N = 100;
#pragma unroll
for(int i = 0; i < N; ++i) {
  // Loop body
 }
// Equivalent to:
//i=0;
// Loop body
//i=1;
// Loop body
// ... until i=99
```

Advice

Unrolling is only possible when the number of iterations is known at compile time. This means that the loop counter must be a constant or a variable that is known at compile time.

4.5 Vectorization

CPUs have SIMD (Single Instruction, Multiple Data) instructions that can perform the same operation on multiple data elements simultaneously. The largest SIMD registers are 512 bits wide (64 bytes), which can hold 16 floats or 8 doubles. This potentially allows for a 16x speedup in some operations.
There are SIMD instructions for a wide range of operations, such as addition, multiplication, and bitwise operations.

Advice

Writing code that takes advantage of SIMD instructions explicitly is quite complicated and low level. Furthermore, targeting a specific SIMD instruction set will make your code non-portable, since different CPUs have different SIMD instruction sets.
Instead, you should rely on the compiler to vectorize your code. Modern compilers are quite good at this, and they can automatically vectorize loops and operations when possible.
You should keep in mind this and try to make things easy for the compiler. For instance, the compiler will probably vectorize this loop into a single instruction:

Source code

```
std::array<float, 16> a, b, c;
// Fill a and b
// This whole loop can be written in a single AVX512 instruction
#pragma unroll
for(int i = 0; i < 16; ++i) {
  c[i] = a[i] + b[i];
 }
```

4.6 Branch prediction

Modern CPUs have a branch predictor that tries to guess the outcome of conditional statements. If the prediction is correct, the CPU can continue executing instructions without waiting for the condition to be evaluated. If the prediction is incorrect, the CPU has to discard the instructions it executed and start over.

Advanced

The branch predictor has been the target of several attacks in the past, such as the Spectre and Meltdown vulnerabilities. These attacks exploit the branch predictor to leak information from the CPU.

We can leverage the branch predictor to improve performance by trying to make our code "predictable". There are certain patterns that the branch predictor can recognize and predict correctly, such as:

- TTTTTTTT: A condition that is always true (or false).
- TFFFFFFF: A condition that is false (or true) most of the time.
- TFTFTFTF: A condition that alternates between true and false.

5 Profiling techniques

5.1 GDB poor man‘s sampling profiler

GDB can be used as a simple sampling profiler. This is not as powerful as a full-fledged profiler, but it can give you a rough idea of where your code is spending most of its time.
The idea is simple, you run your code through gdb and randomly pause it with Ctrl+C. You then check the call stack (with the bt command) to see where the code happened to be. Then resume the execution (with the c command) and repeat this process a few times.
Statistically, you will get a rough idea of where your code is spending most of its time. This is not very

precise, but it is a quick and dirty way to get an idea of where to focus your optimization efforts. Works best with long-running programs in which there is a clear single bottleneck.

5.2 Benchmarking with std::chrono

The `std::chrono` library can be used to measure the time taken by a piece of code. This can be used to benchmark different implementations of the same algorithm or to measure the performance of different parts of your code.
I normally use a function pretty similar to this one to benchmark a particular function:

Source code

```
template <typename Foo> auto bench(int ntest, Foo foo) {
  using namespace std::chrono;
  // Warm up the CPU cache by running the function a few times
  //  without measuring
  for (int i = 0; i < 10; i++)
    foo();
  auto start = high_resolution_clock::now();
  // Measure the time taken by the function by running it ntest times
  for (int i = 0; i < ntest; i++)
    foo();
  auto end = high_resolution_clock::now();
  auto us = duration_cast<microseconds>(end - start).count();
  return us / (ntest * 1.0);
}
```

This function takes a function as an argument and runs it ntest times, measuring the time taken by the function. It returns the average time taken by the function in microseconds. We can use it with a lambda function:

Source code

```
void foo(int some_arg){
  // This is a function to measure
}
//... In main:
int var = 42;
auto time = bench(1000, [&](){foo(var);});
std::cout << "Time taken: " << time << " us\n";
```

5.3 Profiling with gperftools

Google performance tools (gperftools) is a set of tools for profiling and optimizing C++ applications. It includes a CPU profiler, a heap profiler, and a heap checker. You can install it via conda.
In order to use it you need to link your code with the "profiler" library:

Source code

```
$ g++ -lprofiler -o my_program my_program.cpp
```

Then you can run your code with the CPUPROFILE environment variable set to the name of the output file:

Source code

```
$ CPUPROFILE=profile.prof ./my_program
```

This will generate a file called "profile.prof" that you can analyze with the pprof tool:

Source code

```
$ pprof ./my_program profile.prof
(pprof) top
(pprof) web
```

The "top" command will show you the functions that are taking the most time, and the "web" command will open a web browser with a graphical representation of the profile. There are many more commands available, and you can get help by typing "help" in the pprof shell.

6 Exercises

Go over the programs that came with this lesson and experiment with them. Make sure you understand what is happening and why in each case. Try to modify the code to see how it affects the performance. You will notice that some apparently innocuous changes can have a big impact on the performance of your code.