Tute 5: Optimisations & Lambdas

Review:Lambdas & Profiling

What is a lambda function?

Note: this is a modern C++ feature.

A lambda function is an anonymous function, which means it has a definition but no declaration. C++ also supports closures, which means the lambda function can be stored in a variable for future use. They are often passed as arguments to other named functions or classes, or used in place when a function is required for simple/repetitive work without the need for writing a separate function declaration. Lambda functions have four parts:

```
[capture list](parameters) -> return type { function body }
```

eg:

```
[&z](int x, int y) -> int { return x + y; }
```

The capture list can be used in six different ways:

```
[]      //no variables defined. Attempting to use any external variables in the lambda is an error.
[x, &y] //x is captured by value, y is captured by reference
[&]     //any external variable is implicitly captured by reference if used
[=]     //any external variable is implicitly captured by value if used
[&, x]  //x is explicitly captured by value. Other variables will be captured by reference
[=, &z] //z is explicitly captured by reference. Other variables will be captured by value
```

What is profiling?

Profiling is a form of dynamic program analysis (dynamic meaning at runtime) that is used to measure memory usage and/or performance of different parts of a program. It can be used to identify bottlenecks (ie, which functions take the most execution time) so that efforts to optimise can be focussed in those areas as optimisations in non-bottleneck areas will lead to smaller performance gains.

To the tutor: I want you to demonstrate to them how to use qcachegrind/kcachegrind to get useful graphing data on the performance of a program.

Programming Concepts: Lambdas & Profiling

How can a lambda function be passed to a function/method?

The function template can be used:

```
void foo(std::function<void()> t)
{
	t();
}

foo([]() { std::cout << “Lambda\n”; });
```

Or if the return type/arguments are unknown, templates can be used instead:

```
template <typename T>
void foo(T t)
{
	t();
}

foo([]() { std::cout << “Lambda\n”; });
```

Similarly, the easiest way to store a lambda function is using the auto keyword:

```
auto func = []() { std::cout << “Lambda\n”; };
foo(func);
```

Although the type of the variable will still be a std::function.

How can we time different parts of a program to determine what needs to be optimised?

Note: this is a modern C++ feature.

Without using external profiling tools, the chrono header from the standard library can be used to time parts of a program's execution with sub-millisecond accuracy, eg:

```
#include <chrono>

std::chrono::time_point<std::chrono::system_clock> lastTime;
…
auto thisTime = std::chrono::system_clock::now();
std::chrono::duration<float> deltaTime = thisTime - lastTime;
std::cout << deltaTime.count() * 1000.0f;
```

Compilation: Optimisation

What is the difference between the compile flags O1/O2/O3?

- -O1 enables a subset of gcc optimisation flags, usually flags that would not significantly increase compilation time.
- -O2 enables a larger subset of optimisation flags, including those that increase compilation time, however usually not optimisations that would significantly increase executable size.
- -O3 enables almost all optimisation flags, which includes flags that increase executable size. Note that this flag can significantly alter generated code, which may prevent gdb from giving accurate debugging information. If debugging information is needed with the -g flag, -O2 optimisation should normally be used instead.

What are loop unrolling and vectorisation?

Loop unrolling is not enabled by any O level flag and is instead enabled with -funroll-loops which unrolls loops whose number of iterations can be determined at compile time, while -funroll-all-loops will also unroll loops whose number of iterations is uncertain. Unrolling of loops can be done manually:

```
for (int i=0; i<n; i++)
{
        sum += data[i];
}
…
for (int i=0; i<n; i+=4)
{
        sum1 += data[i+0];
        sum2 += data[i+1];
        sum3 += data[i+2];
        sum4 += data[i+3];
    }
    sum = sum1 + sum2 + sum3 + sum4;
```

The potential advantage of this is that if there is a cache miss or branch prediction failure (if the code has branches) or any other stall in on calculation, the other three calculations don't have to wait for the stall and can still execute when using a super-scalar (parallel) CPU or on a CPU that has a SIMD instruction set that can perform the 4 calculations simultaneously.

Loop vectorisation is related, in that a CPU using SIMD can perform multiple calculations (in the above case, four) at once inside a loop, instead of one at a time. Loop vectorisation is enabled when using the -O3 flag. Note that vectorisation or unrolling will not necessarily improve performance, and in some rare cases can decrease performance, for example, some iterative sorting implementations may perform fast with -O2 than -O3.

Errors: None this week

Exercises:

Use a lambda function to sort a vector of ints in reverse sorted order.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char **argv)
{
    std::vector<int> v = {1, 10, 5, 6, 2, 8, 12, 7, 19, 14};
    std::sort(v.begin(), v.end(), [](int x, int y) { return x > y; });

    for (auto i : v)
        std::cout << i << " ";
    std::cout << "\n";
    return 0;
}
```