TDT4200 PS 4 - Profiling and optimizations

A. C. Elster, B. Gotschall and Z. Svela

Due Monday, Oct 7th, 2019

Introduction

This Problem Set is evaluated ass Pass/Fail, and is mandatory for taking the exam.

This assignment will be focused on profiling and optimizing serial code. You will use the profiling tool Valgrind to locate bottlenecks and use this information to optimize them using techniques learned in this course. Your target will be a Mandelbrot set renderer, which can generate very colourful and diverse pictures of complex numbers such as the one in Figure 1.

Mandelbrot set

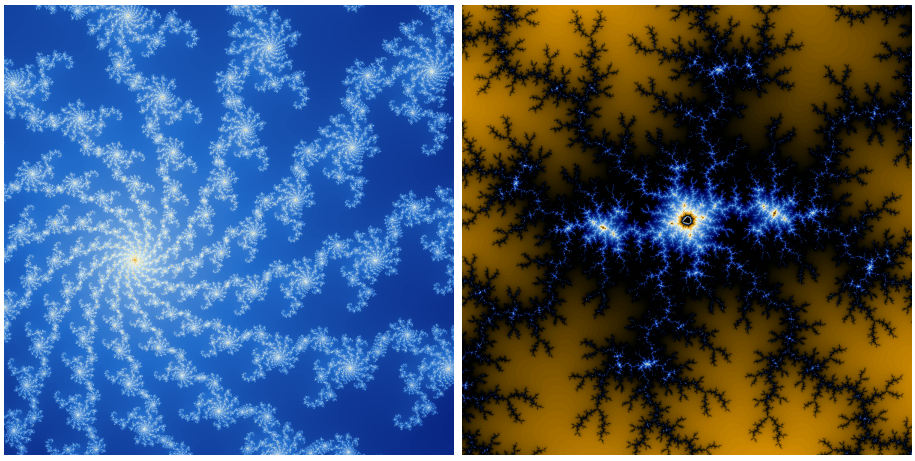

Figure 1: Examples pictures from the Mandelbrot set

The Mandelbrot set is a set of complex numbers for which an iterative function does not diverge. The Mandelbrot set calculation itself is relatively easy as the following definition show.

A complex number c is a member of the Mandelbrot set ($c \in M$) if the following applies:

$$z_0 = 0 \tag{1}$$

$$z_{n+1} = z_n^2 + c \tag{2}$$

$$\left| \limsup_{n \to \infty} z_n \right| \leq 2 \tag{3}$$

If we'd like to draw an image of this set, we can test for each pixel whether it is a member of the set. Unfortunately, doing this accurately requires an infinite number of iterations. As you might imagine, this is not feasible on any hardware. We'll therefore limit n to some value n_{max}.

The needed number of iterations to determine whether a point is a member of the Mandelbrot set is also called the *dwell* value. Consequently, n_{max} is called the maximum dwell value. In our case, we convert a pixel coordinate to a complex number c, and test whether it is a member of the set (where the x and y coordinate become the real and imaginary parts, respectively).

If we were to plot only membership of the Mandelbrot set, the result would be a black and white image. However, we can abuse the dwell value to also assign colours to the complex numbers. The colour mapping of the dwell values is mostly eye candy. Colours are chosen arbitrarily, and can be very different across implementations.

The amazing property of the Mandelbrot set are the rich variety in forms and shapes that can be found by zooming deeper and deeper into the set. The zooming factor is only limited by the hardware and implementation of the renderer, which is in our case the precision of the double precision floating point type. However, the provided renderer allows you to easily reach a zoom factor of 7.5×10^{14} %.

About Valgrind

Valgrind is a set of profiling tools that can help you identify bugs and potential optimizations in your C code. We will be using two of these tools, Cachegrind to identify cache misses and Callgrind to identify which functions the program occupies most of the time. Both will help us locate bottlenecks in our code.

A quick start guide to Valgrind can be found here, and on their web pages there is also documentation on Cachegrind and Callgrind. When these run your code, they will produce a file with profiling information called ⟨tool-name⟩.out.⟨PID⟩ (e.g. "cachegrind.out.4201"). In order to make more sense of this output, we will be using kcachegrind with which the generated profiling information can graphically visualized (see Figure 2).

About the code

The code handed out calculates and visualizes the Mandelbrot set, a subset of which was the image manipulated in assignment 1-3. Your focus will be the

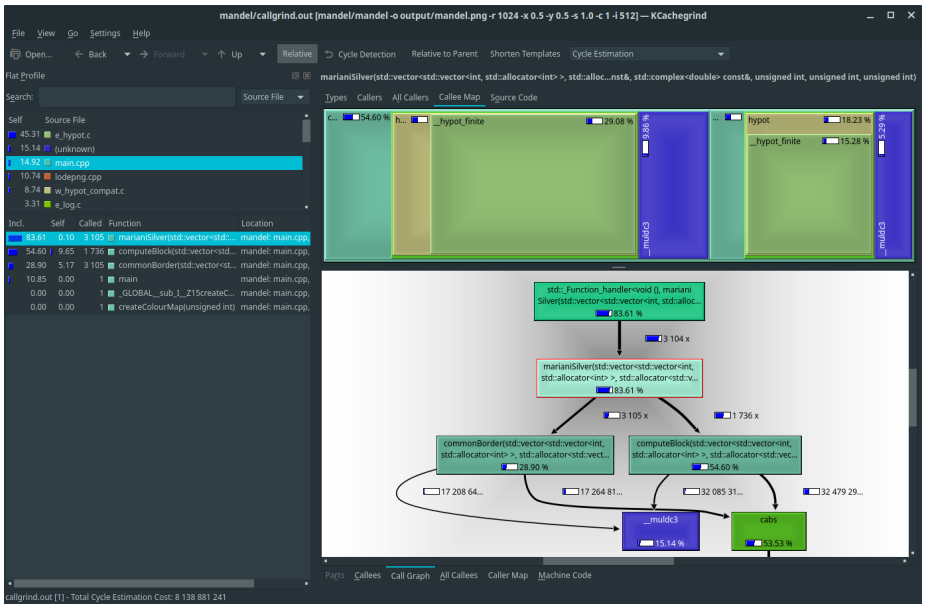

Figure 2: Example for an callgrind output visualized using kcachegrind.

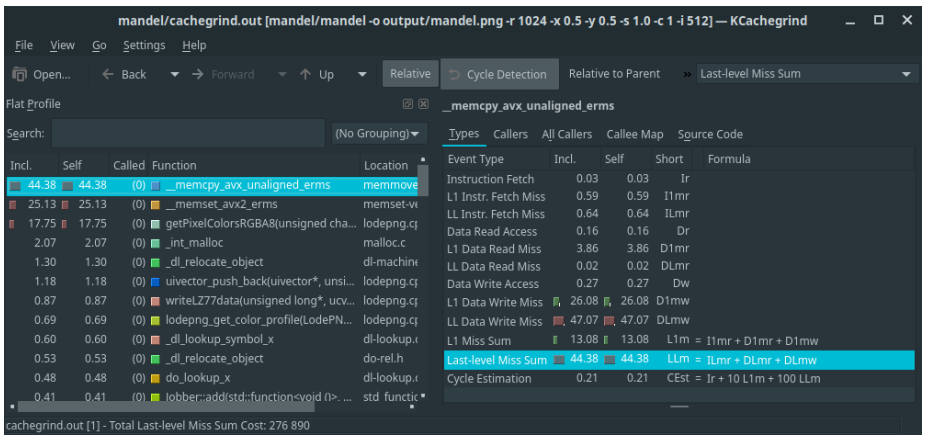

Figure 3: Example for an cachegrind output visualized using kcachegrind.

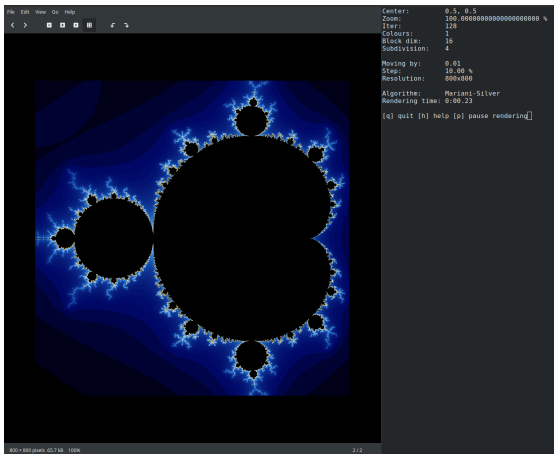

Figure 4: xviewer side by side with mandelNav

file called "main.c", which you will optimize after profiling it. The renderer support various parameters to control the view, maximum iterations, resolution and colours. Try around how to use the parameters and how they impact the rendered image. You are also provided with an interactive shell script called mandelNav, which controls the parameters of the renderer with single button commands. Try it out by opening a terminal and execute `mandelNav.sh` and then open an image viewer like xviewer (or any other image viewer that automatically updates the image if the file changes) on the rendered image at 'output/mandelnav.bmp'. As soon as you interact with mandelNav, an new image gets rendered and displayed by your image viewer. It is recommended to put mandelNav and the image viewer side by side as seen in Figure 4.

The Makefile contains most of the functionality you will need to time your execution, profile the application with cachegrind and callgrind and open mandelNav. An overview about its functionality can be printed on your terminal by running `make`. You can control the different options for example by executing `make RES=2000 I=512 call`.

Preparation

1. Get familiar with the code. Running `make` will print options for both compiling and running the program. Look quickly through the "main.c" file to see the program flow.

2. Try rendering different images and get a feeling for the parameters. Your might use mandelNav for this which also displays the current renderer call with the correct parameters. Get a feeling for the impact of the resolution and iteration number on rendering time.

3. You need to have the following tools installed on your computer (lab machines have them already installed):

   (a) valgrind

   (b) kcachegrind (optional, but recommended)

   (c) dot (optional, but recommended)

   You can install them on a Ubuntu like system with the following command: `sudo apt-get install valgrind kcachegrind graphviz`. If you skip kcachegrind you have to deal with plain text terminal output for profiling your application.

NB!: You must not use compiler optimizations for ALL tasks (-O0). It is also recommended to include debug information (-g). The provided Makefile sets those compiler flags already for you.

Task 1: Baseline measurement

a) **Report:** Measure the runtime of your application by using `time` to execute it or simply running `make time`. Find a set of reasonable parameters (iterations, resolution, x, y, colours) to take the baseline. Always compare in future tasks to this baseline using the same parameters.

b) **Report:** Alter the resolution and iteration number of your baseline parameters and give a rough speedup/slowdown on execution time. Does the position in the Mandelbrot set and zoom level have an impact on this? Give a **short** justification.

Task 2: Profiling

Use Valgrind to identify important bottlenecks.

a) Run the program using Valgrind and choose Cachegrind as the tool. You can do this either manually as described in the tool documentation or execute `make cachegrind`.

b) **Report:** identify at least two positions in the application which show a bad utilization of cache ressources. Document on how bad they behave for your baseline parameters.

c) Run the program using Valgrind and choose Callgrind as the tool. You can do this either manually as described in the tool documentation or execute `make callgrind`.

d) **Report:** Identify 4 functions in the application which are responsible for most of the runtime and document their share of the runtime.

e) **Report:** Measure one of the functions identified in d) by instrumenting the code with the function `clock_gettime` from the `time.h` library and report its total runtime in seconds with a millisecond resolution (with 3 digits after comma) when executed with your baseline parameters.

NB!: Although Callgrind is a very useful tool to get a first impression of spent time in your application, you should consider inserting timing statements in your code to measure certain code regions. This is not only faster but also more precise and helps you in the optimization task.

Task 3: Optimization

Now we will perform optimizations on our code based on the profiling information. Relevant techniques you should remember are function inlining, loop reordering, cache utilization and stack and heap management. In order not to lose information from the profiling, note the positions of the bottlenecks before you move code around. This will make it easier to measure the effect of your optimizations.

a) Using the findings from running Cachegrind, perform optimizations to improve cache performance. Hint: Cachegrind provides information about the type of cache miss, which is useful when employing optimization techniques.

b) **Report.** Re-run your modified code with Cachegrind. Document where and how you optimized your code and what effect it had on cache performance.

c) Now use the findings from Callgrind and perform optimizations on the documented bottlenecks. Hint: One of your goals is to avoid unnecessary function calls, but mostly to ensure that the most frequently used functions and source lines are as efficient as possible. Think also at improving data handling and calculations (e.g. passing parameters as values or as references).

d) **Report.** Re-run and time your application. Document your optimizations and the speedup achieved. You do not have to rerun Callgrind every time as it is itself a very time consuming profiling technique. Instead rely on external and internal time measurements (e.g. use time in the terminal, and/or the time.h library with code instrumentation).

Deliverables

The documentation must be delivered as a PDF file together with the source code packed in a ZIP file and submitted in time on blackboard. The code must be documented and in a runnable state. Do not use any libraries apart from standard libraries or those provided. Do not change the calling conventions (parameters) of the application.

The following tasks require documentation (if solved):

- 1a, 1b, 2b, 2d, 2e, 3b, 3d

Do not include binaries or object files in your delivery and don't include any BMP image files.