EEE3095/6S: Practical 1B

STM32F0 Performance Benchmarking and Profiling

July 30, 2025

1 Overview

In this practical we will explore benchmarking the computing performance of the STM32F0 using a well known benchmark called Mandelbrot set.

What is the Mandelbrot set?
Mandelbrot set is a 2D set that is defined in the complex plane as the complex numbers for which the function $f(z) = z^2 + c$ does not diverge to infinity when iterated starting at $z = 0$ i.e if the absolute value of z remains bounded (typically below 2) after many iterations, the point c is in the Mandelbrot set.

By iterating this simple mathematical function, it reveals complex patterns that challenge computational capabilities, testing floating-point arithmetic, memory management, and parallel processing across different programming languages.

As you have covered in your introductory lectures benchmarking and performance profiling is a critical aspect of embedded systems due to the low computation capabilities. In this practical you will get to implement a smaller version of the Mandelbrot set albeit no plotting of the actual image will be required here, as we are running the code on a "headless" system.

For more information on the Mandelbrot set read and a sample Mandelbrot image is shown in Figure 1:

1. Resource1
2. Resource2
3. Resource3

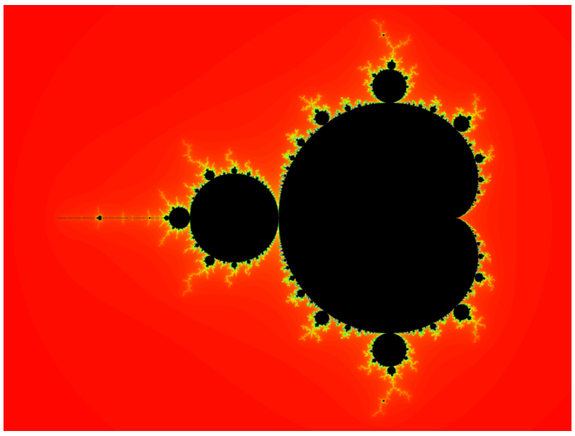
Figure 1: Sample Mandelbrot set image

2 Outcomes and Knowledge Areas

In this practical, you will profile the performance of a simple Mandelbrot run on the STM32F0. After this practical, you should be comfortable profiling embedded code in terms of execution time and simple checksum.

You will learn the following aspects:

- Profile code execution time in an embedded system
- Profiling memory through a simple checksum

3 Deliverables

For this practical, you must:

- Develop the code required to meet all objectives specified in the Tasks section
- Push your completed code to a shared repository on GitHub
- Write a short report documenting your code, GitHub repo link, and a brief description of the implementation of your solutions. This must be in PDF format and submitted on Gradescope with the naming convention(**If you do not adhere to the naming convention, there will be a penalty**):

  EEE3096S 2025 Practical 1B Hand-in STDNUM001 STDNUM002.pdf

  Check the Appendix for Report Structure 6.
- Your practical mark will be based both on your demo to the tutor for Prac 1A as well as your short report. Both you and your partner will receive the same mark.

4 Getting Started

The procedure is as follows:

1. Clone or download the Git repository(The practical folder of interest is Practical1/Practical_1B:
   `git clone https://github.com/EEE3096S-UCT/EEE3096S-2025.git`
2. - Open **STM32CubeIDE**, then navigate through the menus:

     `File` → `Import` → `Existing Code as Makefile Project`

   - Click `Next`.
   - In the dialog, click `Browse...` and select your project folder.
   - Under **Toolchain**, choose `MCU ARM GCC`.
   - Click `Finish`.

   Note: This IDE provides a GUI to set up clocks and peripherals (GPIO, UART, SPI, etc.) and then automatically generates the code required to enable them in the main.c file. The setup for this is stored in an .ioc file, which we have provided in the project folder if you would ever like to see how the pins are configured.**However, it is crucial that you do NOT make/save any changes to this .ioc file as it would re-generate the code in your main.c file and may delete code that you have added.**
3. In the IDE, navigate and open the **main.c** file under the Core/src folder, and then complete the Tasks below.

   Note: All code that you need to write/add in the main.c file is marked with a "TODO" comment; **do not edit any part of the other code that is provided**.

5 Tasks

1. Define variables needed:
   - **Image dimensions** that will be required for testing i.e square images of dimensions (128, 160, 192, 224, 256)
   - A **global checksum** variable to hold the checksum returned from the mandelbrot function
   - A **global start_time** variable
   - A **global end_time** variable

   NOTE: The variables that are to be defined globally should be defined as such as this is critical when you run the code to be able to view the values without connecting serially to a PC as we will be using the debugger view and live expressions feature of the STM32CUBE IDE to view the variable values.

2. Complete the `uint64_t calculate_mandelbrot_fixed_point_arithmetic(int width, int height, int max_iterations)` function to calculate the Mandelbrot set and return the checksum. The pseudocode you are supposed to use is given below in 1 or alternatively you can check the python code provided(`Mandelbrot.py`). You are specifically required to used fixed point arithmetic i.e only use of integers(A recap of fixed point arithmetic is given in Appendix 6):

Algorithm 1 CalculateMandelbrot

```
Require: width, height, maxIter
Ensure: checksum
 1: checksum ← 0
 2: for $y \leftarrow 0$ to height − 1 do
 3:     for $x \leftarrow 0$ to width − 1 do
 4:         $x_0 \leftarrow \frac{x}{\text{width}} \times 3.5 - 2.5$
 5:         $y_0 \leftarrow \frac{y}{\text{height}} \times 2.0 - 1.0$
 6:         $x_i \leftarrow 0, \quad y_i \leftarrow 0$
 7:         iteration ← 0
 8:         while iteration < maxIter and $x_i^2 + y_i^2 \leq 4$ do
 9:             temp ← $x_i^2 - y_i^2$
10:             $y_i \leftarrow 2\,x_i\,y_i + y_0$
11:             $x_i \leftarrow$ temp + $x_0$
12:             iteration ← iteration + 1
13:         end while
14:         checksum ← checksum + iteration
15:     end for
16: end for
17: return checksum
```

   NOTE: The maximum number of iterations is set as **100** in the macros section of the code and kindly do not change it.

3. Call the `uint64_t calculate_mandelbrot_fixed_point_arithmetic(int width, int height, int max_iterations)` and time it using the `start_time` and `end_time` variables. There are various ways to time it, you can use `HAL_Get_Tick` or standard-C clock library.

   NOTE: Take note of the execution time and checksum value and rerun the code for the other image dimensions. To run the code and be able to view the variable values follow the instructions below:

   (a) Click the debug button(bug icon)

   (b) Once in the debug perspective on the left hand pane select live expressions and add the variables you intend to record i.e execution time, checksum

   (c) To run the code click the resume button and wait till the code runs to completion when you see the live expressions turn to pink with values for that run

4. Complete the `uint64_t calculate_mandelbrot_double(int width, int height, int max_iterations)` function to calculate the Mandelbrot set and return the checksum. The only difference from step 2 is now you are to use doubles instead of using fixed point arithmetic. Then redo step 3 using this new function and record the execution times and checksum for all the image dimensions.

5. Run the python code to get the reference checksum to compare the values you got from the STM32F0. You do not have to edit the python code. Run it as it is and it will generate a log file with the reference checksums for each respective image dimensions and also a plot of the corresponding Mandelbrot image saved in a png file. We expect a tolerance of about 1% or less for the checksum.

   NOTE: You need numpy installed for the python code to run. If you do not have numpy installed simply run the command `pip install numpy` on your terminal. All the results are stored in a folder called Mandelbrot_results in the directory from which you ran the code from.

6 Questions to think about

1. Are you getting the exact sum values between the STM32 and the PC?If not why do you think that is the case?
2. Which one is more accurate and faster in execution: Fixed point arithmetic or use of doubles and why?
3. What would happen if we just used integers straight on without using fixed point arithmetic?
4. Will there be a difference if we were to use floats instead of doubles?
5. How does max_iter affect execution time?
6. Estimate if we were to store the image pixels as done in python code but on the STM32F0 , what is the largest image we can store?
7. Do you think a more powerful STM32 board such as STM32F4 will perform better than the STM32F0?

Appendix A: Report Structure

Section	Description
Introduction	Briefly introduce the aim and objectives of the practical and summarise your work briefly.
Methodology	Detail the steps undertaken and methods used to achieve the practical task in a logical and coherent manner.
Results and Discussion	Present the results and discuss their significance.
Conclusion	Summarise the work you did for the practical and any improvements you could make to your implementation.
AI Clause	In one paragraph discuss how you used LLMs while working on the practical and if you did find them useful in an embedded systems programming context.

Appendix B: Fixed-Point Arithmetic Recap

1. **Choose a scale factor** Select S such that $S \times x_{\max}$ fits in your integer type. For 64-bit integers, a common choice is
$$S = 10^6.$$

2. **Convert real values** Multiply each real number by S:
$$1.5 \longrightarrow 1.5 \times 10^6 = 1\,500\,000, \quad 2.0 \longrightarrow 2.0 \times 10^6 = 2\,000\,000.$$

3. **Addition & subtraction** Perform as usual on scaled integers, then divide by S:
$$1.5 + 2.0 = 1\,500\,000 + 2\,000\,000 = 3500000 \quad \big(\text{represents } 3.5\big).$$

4. **Multiplication** Multiply the scaled integers and then divide by S to avoid overscaling:
$$\frac{(1\,500\,000)\,(2\,000\,000)}{S} = 3\,000\,000 \quad \big(\text{represents } 3.0\big).$$

5. **Division** First scale the dividend, then divide:
$$\frac{(1\,500\,000)\,S}{2\,000\,000} = 750\,000 \quad \big(\text{represents } 0.75\big).$$

6. **Convert back** Divide the final integer result by the scale factor:
$$\frac{3.0 \times 10^6}{10^6} = 3.0.$$