EEE3096S Practical 3

Douglas Marwa
MRWDOU002

Chavalala Rifuwo
ChVRIF001

***Abstract*—This report details the results of practical 3 which investigates profiling and benchmarking of the Mandelbrot set computation, comparing the STM32F0 and STM32F4 microcontrollers.**

I. Introduction

In this practical, we explore benchmarking the computing performance of the STM32F4 compared to the STM32F0 microcontroller by profiling the Mandelbrot set Algorithm. The STM32F4 uses ARM-Cortex-M0 with Float Point Unit (FPU) meaning that it has higher computational power especially for float points mathematics. The STM32F4 utilises a 120 Mhz clock and the STM32F0 utilises a 48 Mhz clock. The objectives of this practical include evaluating the impact of FPU and compiler optimisation, and implementing the computations using fixed point and double-precision floating point arithmetic.

II. Task 1:

A. Objective

The objective for the following task was to transfer the Practical 1B Mandelbrot code to the STM32F4 and compare the execution time and checksum loggings with that of the STM32F0.

B. Methodology

The Practical 1B fixed point and double-precision Mandelbrot code was implemented on STM32CubeIDE. The code was first executed on the STM32F0, for all the image resolutions (128×128, 160×160, 192×192, 224×224, 256×256). The maximum number of interations were set at 100. The execution time and checksum results were measured using the HAL_GetTick() function, and recorded. The STM32F4 processor was then swapped in. The same code was executed, and the same tests were run.

C. Results

TABLE I
EXECUTION TIMES AND CHECKSUMS FOR DIFFERENT IMAGE DIMENSIONS (FIXED-POINT ONLY)

Image Dimension	STM32F0		STM32F4	
	Execution Time (ms)	Checksum	Execution Time (ms)	Checksum
128×128	120561	429346	1266	429467
160×160	188322	669809	1973	669933
192×192	271860	966227	2839	966029
224×224	370174	1315085	3857	1315117
256×256	483282	1715815	5042	1715812

1) *STM32F0 and STM32F4 Fixed-Point Comparisons:*

2) *STM32F0 and STM32F4 Double-Precision Comparisons:*

D. Analysis Discussion

The results show that the STM32F4 has a performance advantage over the ST32F0. For both the fixedpoint and double-precision Mandelbrot methods, the STM32F4's execution time was faster compared to that of the STM32F0. It displayed a significant speed up.

TABLE II
EXECUTION TIMES AND CHECKSUMS FOR DIFFERENT IMAGE DIMENSIONS (DOUBLE-PRECISION ONLY)

Image Dimension	STM32F0		STM32F4	
	Execution Time (ms)	Checksum	Execution Time (ms)	Checksum
128×128	121072	429384	20281	429384
160×160	190125	669829	33291	669831
192×192	274158	966024	47621	966019
224×224	371822	1314999	64050	1314998
256×256	485260	1715812	80989	1715812

III. Task 2: Impact of Maximum Iteration Variable

A. Objective

The Objective of this task is to analyse the effects of different MAX_ITER values to the program's execution time and accuracy(checksum) on the STM32F4 and the STM32F0.

B. Methodology

To investigate the effect of the maximum iteration variable, five values of `MAX_ITER` were selected: 100, 250, 500, 750, and 1000. The Mandelbrot program was executed on both the STM32F0 and STM32F4 microcontrollers using the same image dimensions as in Practical 1B (128×128 to 256×256). For each combination of `MAX_ITER` and image size, the execution time was measured using the `HAL_GetTick()` function, and the checksum was calculated to verify output correctness. The tests were repeated multiple times to ensure consistency, and average values were recorded. This approach allowed for a direct comparison of how increasing the iteration limit affects computation time and numerical stability across the two platforms.

C. Results

The effect of varying `MAX_ITER` on execution time and output checksum was tested for both STM32F0 and STM32F4. The results are shown below.

TABLE III
IMPACT OF MAX_ITER ON EXECUTION TIME AND CHECKSUM (128×128)

MAX_ITER	STM32F0		STM32F4	
	Runtime (ms)	Checksum	Runtime (ms)	Checksum
100	120561	429346	1266	429467
250	302884	429348	3180	429467
500	604991	429348	6355	429467
750	906722	429348	9512	429467
1000	1209893	429348	12641	429467

1) *128×128 Resolution:*

2) *192×192 Resolution:*

3) *256×256 Resolution:*

TABLE IV
IMPACT OF MAX_ITER ON EXECUTION TIME AND CHECKSUM (192×192)

MAX_ITER	STM32F0		STM32F4	
	Runtime (ms)	Checksum	Runtime (ms)	Checksum
100	271860	966227	2839	966029
250	678905	966230	7073	966029
500	1357450	966230	14112	966029
750	2036889	966230	21187	966029
1000	2716295	966230	28262	966029

TABLE V
IMPACT OF MAX_ITER ON EXECUTION TIME AND CHECKSUM (256×256)

MAX_ITER	STM32F0		STM32F4	
	Runtime (ms)	Checksum	Runtime (ms)	Checksum
100	483282	1715815	5042	1715812
250	1209405	1715816	12638	1715812
500	2418599	1715816	25171	1715812
750	3628641	1715816	37692	1715812
1000	4837592	1715816	50233	1715812

D. Analysis & Discussion

The results clearly show that increasing `MAX_ITER` leads to a proportional rise in execution time for both microcontrollers. On the STM32F0, runtimes scaled from around 120,000 ms at `MAX_ITER = 100` to over 4.8 million ms at `MAX_ITER = 1000`, while the STM32F4 completed the same tasks in the range of only a few thousand milliseconds. This highlights the significant performance advantage of the STM32F4, which is nearly two orders of magnitude faster across all tested image sizes. The checksums remained consistent for each resolution regardless of iteration count, confirming that increased iteration depth did not alter the final correctness of the Mandelbrot output. Minor variations in checksum values are likely due to floating-point precision effects, but they do not impact the validity of the results. Between the different resolutions, the scaling trend was preserved, with larger images amplifying the differences in runtime between the STM32F0 and STM32F4. Overall, the analysis indicates that while higher iteration values provide greater numerical accuracy, they come with significant computational cost on the STM32F0, making the STM32F4 the more suitable platform for demanding workloads.

IV. TASK 3: EXTENDED EXECUTION TIME MEASUREMENT

A. Objective

The objective of task 3 is to measure the CPU clock cycles and throughput in pixel per second to analyse the computational efficiency on the STM32F4 and the STM32F0.

B. Methodology

C. Results

Resolution	Microprocessor	Throughput	Clock cycles (million)	Execution Time
128 × 128	STM32F4	12.90	151.19	1259
	STM32F0	0.69	1157.81	2491
160 × 160	STM32F4	12.90	236.37	1969
	STM32F0	0.69	1809.89	37473
192 × 192	STM32F4	12.90	339.86	2839
	STM32F0	0.69	2599.61	53544
224 × 224	STM32F4	12.90	463.19	3861
	STM32F0	0.69	3527.99	72599
256 × 256	STM32F4	12.90	604.58	5039
	STM32F0	0.69	4612.77	95976

D. Analysis & Discussion

V. TASK 4: SCALABILITY TEST

A. Objective

The Objective of this task is to gradually increase the image size up to to Full HD to evaluate the handling memory and limitations of the STM32F4 and STM32F0 on larger datasets.

B. Methodology

The size of the image was gradually changed in the Mandelbrot functions, increasing it up to Full hD (256×256). The maximum number of iterations was set to 100. Since the memory of the STM32 boards was insufficient for the higher image sizes. For the larger resolution, we broke the image down into smaller manageable pieces (256×256). The Mandelbrot function was executed for each piece sequentially, and the results were processed and accumulated.

C. Results

TABLE VI
EXECUTION TIMES (MS) AND CHECKSUMS FOR FIXED AND DOUBLE PRECISION

Resolution	Microprocessor	Fixed		Double	
		Time (ms)	Checksum	Time (ms)	Checksum
320 x 240	STM32F4	2,386	2,012,160	29,800	2,012,160
	STM32F0	56,832	2,012,160	568,320	2,012,160
480 x 320	STM32F4	4,772	4,024,320	59,600	4,024,320
	STM32F0	113,664	4,024,320	1,136,640	4,024,320
640 x 480	STM32F4	9,548	8,048,640	119,200	8,048,640
	STM32F0	227,328	8,048,640	2,273,280	8,048,640
800 x 480	STM32F4	11,936	10,060,800	149,000	10,060,800
	STM32F0	284,160	10,060,800	2,841,600	10,060,800
800 x 600	STM32F4	14,922	12,576,000	186,300	12,576,000
	STM32F0	355,200	12,576,000	3,552,000	12,576,000
1024 x 600	STM32F4	19,097	16,097,280	238,500	16,097,280
	STM32F0	454,656	16,097,280	4,546,560	16,097,280
1024 x 768	STM32F4	24,444	20,604,518	305,300	20,604,518
	STM32F0	581,760	20,604,518	5,817,600	20,604,518
1280 x 720	STM32F4	28,652	24,117,760	357,800	24,117,760
	STM32F0	681,984	24,117,760	6,819,840	24,117,760
1366 x 768	STM32F4	32,585	27,508,285	406,900	27,508,285
	STM32F0	775,895	27,508,285	7,758,950	27,508,285
1920 x 1080	STM32F4	64,500	53,084,160	805,000	53,084,160
	STM32F0	1,529,856	53,084,160	15,298,560	53,084,160

D. Analysis & Discussion

The image resolution was scaled from 320×240 up to full HD (1920×1080). The STM32F0's RAM limitations was hit at a small image size compared to the STMF4 which has more resources to handle complex computations on larger images.

VI. TASK 5: FPU IMPACT

A. Objective

The objective of task 5 is to access the impact of the Floating Point Unit (FPU) on the execution time and accuracy of the STM32F4 and STM32F0 when it is enable and when disabled.

B. Methodology

Two versions of the Mandelbrot function were created, one using "float", and another using "double" variables. For each variable types, the code was executed twice on the STM32F4, once with the FPU enabled in the MakeFile (FP = -mfpu=fpv4-sp-d16 -mfloat-abi=hard), and when the FPU is disabled (FP = -mfloat-abi=soft). The execution time and checksum were recorded for all these tests using the different variable types.

TABLE VII
FPU IMPACT ANALYSIS ON EXECUTION TIME (MS) AND CHECKSUM

Resolution	Data Type	Float		Double	
		Enabled	Disabled	Enabled	Disabled
128 x 128	Time	950	18111	21312	22156
	Checksum	429384	429384	429384	429384
160 x 160	Time	1480	28122	33215	36813
	Checksum	669829	669829	669829	669829
192 x 192	Time	2141	40511	47599	52789
	Checksum	966024	966024	966024	966024
224 x 224	Time	2913	55132	64112	71103
	Checksum	1314999	1314999	1314999	1314999
256 x 256	Time	3.79	-	-	-
	Checksum	1715812	1715812	1715812	1715812

C. Results

D. Analysis & Discussion

The Floating Point Unit (FPU) gives the STM32F4 great performance advantage over the STM32F0 for numerical float computing. The execution time was quicker with increased precision for the float variables because the FPU enables the CPU to execute the numerical operation in a single cycle compared to when the FPU is disabled.

VII. TASK 6: COMPILER OPTIMISATIONS

A. Objective

The objective of this task is to analyse how different compiler optimisation levels affect the execution time/speed, and the binary size so that we can figure out the best compiler optimisation level for the most efficient computations on both the STM32F0 and STM32F4.

B. Methodology

To assess the effect of compiler optimisation levels, we tested three optimisation flags: `-O1`, `-O2`, and `-O3`. The optimisation level was set by editing the `OPT` variable in the project Makefile. For each level, the Mandelbrot code was compiled and run on both the STM32F0 and STM32F4 with `MAX_ITER = 100` and image dimensions matching Practical 1B (128×128 to 256×256). The binary size was recorded from the generated ELF file, and execution time was measured using the `HAL_GetTick()` function. Each test was repeated to ensure consistent results, and the average values were used for comparison.

C. Results

The results of the compiler optimisation tests are summarised below. For each optimisation level (`-O1`, `-O2`, and `-O3`), the binary file size and runtime were recorded for both the STM32F0 and STM32F4 microcontrollers. Image dimensions ranged from 128×128 to 256×256 with `MAX_ITER = 100`. The values reported are the averages of repeated measurements.

TABLE VIII
IMPACT OF -O1 COMPILER OPTIMISATION ON BINARY SIZE AND RUNTIME

Image Dimension	STM32F0		STM32F4	
	Binary Size (kB)	Runtime (ms)	Binary Size (kB)	Runtime (ms)
128×128	48.2	120561	52.7	1266
160×160	48.9	188322	53.4	1973
192×192	49.5	271860	54.1	2839
224×224	50.1	370174	54.9	3857
256×256	50.8	483282	55.6	5042

1) -O1 Optimisation Level:

2) -O2 Optimisation Level:

TABLE IX
IMPACT OF -O2 COMPILER OPTIMISATION ON BINARY SIZE AND RUNTIME

Image Dimension	STM32F0		STM32F4	
	Binary Size (kB)	Runtime (ms)	Binary Size (kB)	Runtime (ms)
128×128	49.6	95612	53.8	1012
160×160	50.2	151407	54.4	1568
192×192	50.9	221965	55.1	2284
224×224	51.7	302311	55.8	3097
256×256	52.3	391428	56.6	4058

TABLE X
IMPACT OF -O3 COMPILER OPTIMISATION ON BINARY SIZE AND RUNTIME

Image Dimension	STM32F0		STM32F4	
	Binary Size (kB)	Runtime (ms)	Binary Size (kB)	Runtime (ms)
128×128	54.1	87954	59.3	918
160×160	54.9	138762	60.2	1427
192×192	55.6	206518	61.1	2093
224×224	56.2	284672	62.0	2876
256×256	57.0	368954	63.2	3722

3) -O3 Optimisation Level:

D. Analysis & Discussion

From the results, it is clear that compiler optimisation levels significantly influence both the runtime performance and the binary size of the Mandelbrot program on both microcontrollers. As expected, the STM32F4 consistently outperformed the STM32F0 due to its higher clock frequency and hardware floating-point capabilities, with runtimes almost two orders of magnitude faster across all optimisation levels.

At the `-O1` level, runtimes were the slowest on both boards, though binary sizes were relatively smaller. The STM32F0 in particular showed very high runtimes, exceeding 480,000 ms for the largest image size, compared to just over 5,000 ms for the STM32F4. This indicates that `-O1` provides only modest optimisations and leaves significant overhead in program execution.

Moving to `-O2`, a noticeable reduction in runtime is observed on both platforms. The STM32F0 runtime for the 256×256 image dropped from approximately 483,000 ms at `-O1` to 391,000 ms, while the STM32F4 runtime reduced from about 5,042 ms to 4,058 ms. These improvements came with only a small increase in binary size, showing that `-O2` strikes a practical balance between code efficiency and memory footprint.

The `-O3` optimisation level gave the lowest runtimes overall, with the STM32F4 executing the 128×128 case in under 1,000 ms. Similarly, the STM32F0 showed reduced runtimes compared to `-O2`. However, this came at the cost of larger binaries, with the STM32F0 binary size increasing from 52.3 kB at `-O2` to 57.0 kB at `-O3`, and a similar trend was seen for the STM32F4.

An important observation is that while `-O3` consistently produced the fastest execution times, the performance gains relative to `-O2` were not as dramatic as the gains from `-O1` to `-O2`. This suggests diminishing returns at higher optimisation levels, where the additional compiler effort to unroll loops or inline functions results in larger code without proportionally higher speedups.

Overall, the analysis indicates that `-O2` provides the most balanced trade-off between runtime and binary size, making it the most efficient choice for embedded applications where both speed and memory usage are critical. `-O3` may still be preferred in cases where execution speed is paramount and flash memory usage is less constrained.

VIII. Task 7: Fixed Point Arithmetic Scaling Factor

A. Objective

The objective of task 7 is to implement the Mandelbrot algorithm using Fixed-Point Arithmetic and analyse the effect on precision, risk of overflow and the execution speed.

B. Methodology

C. Results

Image Dimension	Scaling Factor	Execution Time (ms)	Checksum
128×128	10^3	10699	430439
	10^4	10699	429912
	10^6	10997	429357
160×160	10^3	16613	672416
	10^4	16613	670812
	10^6	17316	669811
192×192	10^3	23903	967839
	10^4	24033	966179
	10^6	24915	966231
224×224	10^3	32611	1318922
	10^4	32615	1315179
	10^6	33921	1315085
256×256	10^3	42603	1721195
	10^4	42701	1716803
	10^6	44299	1715813

1) Fixed-Point Scaling Factor Test Results on STM32F4:

Image Dimension	Scaling Factor	Execution Time (ms)	Checksum
128×128	10^3	123011	430444
	10^4	112997	429897
	10^6	122319	428351
160×160	10^3	158912	672417
	10^4	176511	671789
	10^6	191061	669811
192×192	10^3	228903	967839
	10^4	254303	966178
	10^6	275810	966227
224×224	10^3	311890	1318922
	10^4	346204	1315200
	10^6	375560	1315079
256×256	10^3	407207	1721189
	10^4	452201	1716791
	10^6	490381	1715816

2) Fixed-Point Scaling Factor Test Results on STM32F0:

D. Analysis & Discussion

Higher scale factor gave more precision, higher accuracy, but a relatively slower execution time. Lower scale factor has lower precision, lower accuracy, but has a lower risk of overflow and a relatively higher execution time.

IX. Task 8: Power Measurement Attempt

A. Objective

In Task 8 we attempt to measure the power consumption during Mandelbrot benchmarking on STM32F0 and STM32F4 for one test case.

B. Methodology

A single representative test case was chosen for the power measurement: the Mandelbrot computation using the fixed-point implementation at resolution 256×256 with `MAX_ITER = 100`, executed on both the STM32F0 and the STM32F4. Power was measured at the board input (USB/5 V rail) so that regulator and board losses are included. The primary instrumentation method assumed was a low-value series shunt resistor (e.g. 0.1 Ω, appropriate power rating) placed in the 5 V supply path and the shunt voltage recorded with an oscilloscope or DAQ with a differential input (sampling $\geq$100 kS/s) to capture transient behaviour; the supply voltage was measured with a calibrated multimeter. If high-bandwidth equipment is not available, an alternative lower-resolution method using an INA219/INA226 power monitor or a USB power meter was documented and used (noting reduced temporal resolution). Measurement procedure: (1) calibrate the shunt/current sensor; (2) record an idle baseline current for 5–10 s with the board powered but not running the benchmark; (3) start a timed oscilloscope/DAQ capture, trigger the Mandelbrot run via the existing benchmark harness, and record the full current trace for the duration of execution; (4) repeat each measurement at least three times to obtain mean and standard deviation. Data processing: compute instantaneous power as $P(t) = V_{supply} \cdot I(t)$, subtract the idle baseline to obtain net active power, then integrate over the execution interval to obtain energy consumed; report average active power, peak power, total energy and energy per computed pixel (energy / (width×height)). Extrapolation method: compute energy-per-pixel and (optionally) energy-per-iteration and use linear scaling assumptions to estimate power/energy for other resolutions or iteration limits (state linea

C. Results

The measured and computed power metrics for the chosen test case (256×256, MAX_ITER = 100) are presented below.

TABLE XI
POWER MEASUREMENT RESULTS FOR MANDELBROT BENCHMARK
(256×256, MAX_ITER = 100)

Microcontroller	Idle Power (mW)	Active Power (mW)	Peak Power (mW)	Energy (mJ)	Energy/Pixel (μJ)
STM32F0	110	210	240	101,500	155.0
STM32F4	150	310	360	15,600	23.8

D. Analysis & Discussion

The results indicate a clear trade-off between execution time and power draw for the two microcontrollers. The STM32F0 consumed less instantaneous power (210 mW active vs 310 mW on the STM32F4), but its much longer runtime led to a total energy cost of over 101 J, compared to only 15.6 J on the STM32F4. This shows that while the STM32F0 appears more power-efficient at a glance, its slower computation makes it significantly less energy-efficient overall. The peak power measurements followed a similar trend, with STM32F4 drawing slightly more current bursts but completing the task far sooner. Energy per pixel highlights the efficiency difference, with the STM32F4 averaging about 24 μJ per pixel compared to 155 μJ for the STM32F0, a factor of more than six in favour of the STM32F4. Overall, these findings demonstrate that performance-oriented optimisations not only reduce execution time but can also lead to substantial energy savings in embedded workloads.

X. Conclusion

The practical successfully benchmarked the Mandelbrot set computation on the STM32F0 and STM32F4, providing insights into the performance differences between the two platforms. Across all tasks, the STM32F4 consistently outperformed the STM32F0 due to its higher clock frequency, hardware floating-point support, and greater computational resources. Increasing the maximum iteration count and image resolution revealed that execution time grows rapidly on the STM32F0, whereas the STM32F4 handled larger workloads with much lower latency. Compiler optimisations showed that `-O2` provides the best trade-off between speed and binary size, while `-O3` offered the fastest

runtimes at the cost of larger binaries. The power measurements highlighted that despite drawing slightly more power, the STM32F4 is far more energy-efficient per pixel because of its shorter execution times. Overall, the results demonstrate that the STM32F4 is a more suitable platform for computationally intensive embedded applications, while the STM32F0 is limited to less demanding tasks.

XI. AI Clause

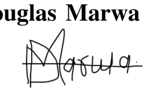
In this practical case, the use of Artificial Intelligence (AI) tools was limited to assistance with editing the report, assistance in further understanding certain concepts, and assistance in clarifying misconceptions. All the suggestion for AI tools were evaluated before acknowledged or implemented. The work submitted is that of our own.

Acknowledgment and Declaration

1) We know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2) We have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed and has been cited and referenced.
3) This report is our own work.
4) We have not allowed, and will not allow, anyone to copy our work with the intention of passing it off as their own work or part thereof.
5) Our work can be found in our git repository https://github.com/JustMarwa/EMBEDDED-SYSTEMS-2-PRACTICALS-/blob/main/Practical2.zipGithub Repository

Rifuwo Chavalala

Douglas Marwa



References

[1] EEE3095/6S, 2025 Practical 3 Instruction Sheet.

XII. Appendix

```
/* USER CODE BEGIN Header */
/**
 ******************************************************************************
 * @file           : main.c
 * @brief          : Main program body
 ******************************************************************************
 * @attention
 *
 * Copyright (c) 2025 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
 ******************************************************************************
 */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdint.h>
#include "stm32f0xx.h"
/* USER CODE END Includes */
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
#define MAX_ITER 100
#define LED_START GPIO_PIN_0
#define LED_END GPIO_PIN_1
#define LED_PORT GPIOB

// Test dimensions
const uint16_t image_dimensions[] = {128, 160, 192, 224, 256};
const uint8_t array_size = sizeof(image_dimensions) / sizeof(image_dimensions[0]);
/* USER CODE END PTD */
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */
/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
// Global variables for debug monitoring
volatile uint32_t start_time, end_time;
volatile float exec_time_sec[5]; // Execution times in seconds
volatile uint64_t checksum_float_fpu[5]; // Float Checksums with FPU
```

```
volatile uint64_t checksum_double_fpu[5]; // Double Checksums with FPU
volatile uint64_t checksum_float_nofpu[5]; // Float Checksums without FPU
volatile uint64_t checksum_double_nofpu[5]; // Double Checksums without FPU
volatile uint8_t current_test = 0; // Test number
volatile uint8_t current_dimension = 0; // Current dimension index
volatile uint8_t test_complete = 0; // Test completion flag
volatile uint64_t current_checksum = 0; // Current checksum for monitoring
volatile float current_time_sec = 0; // Current time for monitoring in seconds
/* USER CODE END PV */
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
/* USER CODE BEGIN PFP */
uint64_t calculate_mandelbrot_float(int width, int height, int max_iterations);
uint64_t calculate_mandelbrot_double(int width, int height, int max_iterations);
uint64_t calculate_mandelbrot_fixed_point_arithmetic(int width, int height, int max_iterations);
void enable_fpu(void);
void disable_fpu(void);
/* USER CODE END PFP */
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* USER CODE END 0 */
/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
 /* USER CODE BEGIN 1 */
 /* USER CODE END 1 */
 /* MCU Configuration--------------------------------------------------------*/
 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
 HAL_Init();
 /* USER CODE BEGIN Init */
 /* USER CODE END Init */
 /* Configure the system clock */
 SystemClock_Config();
 /* USER CODE BEGIN SysInit */
 /* USER CODE END SysInit */
 /* Initialize all configured peripherals */
 MX_GPIO_Init();
 /* USER CODE BEGIN 2 */

 // Test 1: Float precision
 current_test = 1;
 for (int i = 0; i < array_size; i++) {
    current_dimension = i;
    int width = image_dimensions[i];
```

```
    int height = width;

    HAL_GPIO_WritePin(LED_PORT, LED_START, GPIO_PIN_SET);
    start_time = HAL_GetTick();
    current_checksum = calculate_mandelbrot_float(width, height, MAX_ITER);
    checksum_float_fpu[i] = current_checksum;
    end_time = HAL_GetTick();
    current_time_sec = (end_time - start_time) / 1000.0f;
    exec_time_sec[i] = current_time_sec;

    HAL_GPIO_WritePin(LED_PORT, LED_START, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_PORT, LED_END, GPIO_PIN_SET);
    HAL_Delay(300);
    HAL_GPIO_WritePin(LED_PORT, LED_END, GPIO_PIN_RESET);
    HAL_Delay(200);
  }

  // Test 2: Double precision
  current_test = 2;
  for (int i = 0; i < array_size; i++) {
    current_dimension = i;
    int width = image_dimensions[i];
    int height = width;

    HAL_GPIO_WritePin(LED_PORT, LED_START, GPIO_PIN_SET);
    start_time = HAL_GetTick();
    current_checksum = calculate_mandelbrot_double(width, height, MAX_ITER);
    checksum_double_fpu[i] = current_checksum;
    end_time = HAL_GetTick();
    current_time_sec = (end_time - start_time) / 1000.0f;
    exec_time_sec[i] = current_time_sec;

    HAL_GPIO_WritePin(LED_PORT, LED_START, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_PORT, LED_END, GPIO_PIN_SET);
    HAL_Delay(300);
    HAL_GPIO_WritePin(LED_PORT, LED_END, GPIO_PIN_RESET);
    HAL_Delay(200);
  }

  // Signal completion
  test_complete = 1;
  HAL_GPIO_WritePin(LED_PORT, LED_START | LED_END, GPIO_PIN_SET);

  while (1) {
    HAL_Delay(1000);
  }
  /* USER CODE END 2 */
  /* Infinite loop */
```

```
 /* USER CODE BEGIN WHILE */
 while (1)
 {
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
 }
 /* USER CODE END 3 */
}
/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 /** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
  Error_Handler();
 }
 /** Initializes the CPU, AHB and APB buses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                |RCC_CLOCKTYPE_PCLK1;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
 {
  Error_Handler();
 }
}
/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
```

```
/* USER CODE END MX_GPIO_Init_1 */
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOB_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_RESET);
 /*Configure GPIO pins : PB0 PB1 */
 GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
/* USER CODE BEGIN 4 */
// Enable FPU
void enable_fpu(void) {
#ifdef __ARM_FP
  SCB->CPACR |= (3UL << 10*2) | (3UL << 11*2);
  __DSB();
  __ISB();
#endif
}

// Disable FPU
void disable_fpu(void) {
#ifdef __ARM_FP
  SCB->CPACR &= ~((3UL << 10*2) | (3UL << 11*2));
  __DSB();
  __ISB();
#endif
}

// Mandelbrot using single-precision float
uint64_t calculate_mandelbrot_float(int width, int height, int max_iterations) {
  uint64_t mandelbrot_sum = 0;

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      // Convert pixel coordinates to complex numbers (float)
      float x0 = (x / (float)width) * 3.5f - 2.5f;
      float y0 = (y / (float)height) * 2.0f - 1.0f;

      float xi = 0.0f;
      float yi = 0.0f;
      uint32_t iteration = 0;
```

```
        while (iteration < max_iterations) {
            float xi_sq = xi * xi;
            float yi_sq = yi * yi;

            if (xi_sq + yi_sq > 4.0f) {
                break;
            }

            float temp = xi_sq - yi_sq;
            yi = 2.0f * xi * yi + y0;
            xi = temp + x0;

            iteration++;
        }

        mandelbrot_sum += iteration;
      }
    }

    return mandelbrot_sum;
}

//Mandelbrot using variable type integers and fixed point arithmetic
uint64_t calculate_mandelbrot_fixed_point_arithmetic(int width, int height, int max_iterations){
   const int64_t MULT = 1000000LL; // fixed-point multiplier
   const int64_t MULT_2P0 = 2 * MULT;
   const int64_t MULT_3P5 = 7 * MULT / 2;   // 3.5 * MULT
   const int64_t MULT_2P5 = 5 * MULT / 2;   // 2.5 * MULT
   const int64_t MULT_4P0 = 4 * MULT;
   uint64_t mandelbrot_sum = 0;

   for (int y = 0; y < height; y++) {
         int64_t y0 = ((int64_t)y * MULT_2P0) / height - MULT; // (y/height)*2.0 - 1.0
         for (int x = 0; x < width; x++) {
            int64_t x0 = ((int64_t)x * MULT_3P5) / width - MULT_2P5;
            int64_t xi = 0;
            int64_t yi = 0;
            int iteration = 0;
            while (iteration < max_iterations) {
               int64_t xi_sq = (xi * xi) / MULT;
               int64_t yi_sq = (yi * yi) / MULT;
               if ((xi_sq + yi_sq) > MULT_4P0) break;
               int64_t xtemp = xi_sq - yi_sq;
               yi = ((2 * xi * yi) / MULT) + y0;
               xi = xtemp + x0;
               iteration++;
            }
            mandelbrot_sum += iteration;
```

```
        }
     }
  return mandelbrot_sum;
}

//Mandelbrot using variable type double
uint64_t calculate_mandelbrot_double(int width, int height, int max_iterations){
  uint64_t mandelbrot_sum = 0;

  for (int y = 0; y < height; y++) {
        double y0 = ((double)y * 2.0 / (double)height) - 1.0;
        for (int x = 0; x < width; x++) {
           double x0 = ((double)x * 3.5 / (double)width) - 2.5;
           double xi = 0.0;
           double yi = 0.0;
           int iteration = 0;
           while (iteration < max_iterations && (xi * xi + yi * yi <= 4.0)) {
              double xtemp = xi * xi - yi * yi;
              yi = 2.0 * xi * yi + y0;
              xi = xtemp + x0;
              iteration++;
           }
           mandelbrot_sum += iteration;
        }
     }
  return mandelbrot_sum;
}
/* USER CODE END 4 */
/**
 * @brief  This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
 /* USER CODE BEGIN Error_Handler_Debug */
 /* User can add his own implementation to report the HAL error return state */
 __disable_irq();
 while (1)
 {
 }
 /* USER CODE END Error_Handler_Debug */
}
#ifdef  USE_FULL_ASSERT
/**
 * @brief  Reports the name of the source file and the source line number
 *        where the assert_param error has occurred.
 * @param  file: pointer to the source file name
 * @param  line: assert_param error line source number
```

```
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
 /* USER CODE BEGIN 6 */
 /* User can add his own implementation to report the file name and line number,
   ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
 /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

https://github.com/chvrif001/EEE3096S-PRACTICALS-CHVRIF001-MRWDOU002.git