STM32F0 performance benchmarking using Mandelbrot set calculations

Zayaan Lodewyk
LDWZAY001

Ammaarah Hansa
HNSAMM001

***Abstract*— This paper presents benchmarking and performance profiling of the STM32F0 microcontroller using a Mandelbrot set computation task. Both fixed-point arithmetic (using 64-bit integers with a scaling factor of 1,000,000) and double-precision floating-point implementations were evaluated for multiple image resolutions, with execution times and checksums compared to a Python reference implementation. Results indicate that both implementations show comparable performance characteristics, with checksums closely matching the Python reference implementation. These findings highlight the performance characteristics and numerical accuracy achievable on resource-constrained embedded systems.**

Keywords—STM32F0, Mandelbrot, embedded systems, fixed-point arithmetic, floating-point arithmetic, performance benchmarking

Introduction

Benchmarking and profiling are essential for embedded systems design because performance and numerical accuracy often trade off against code size and power. The STM32F0 (Cortex-M0) lacks hardware FPU, so floating-point operations are performed in software and are relatively slow. A useful workload for evaluating compute-bound performance and numerical effects in embedded contexts is the Mandelbrot fractal, which requires repetitive arithmetic and branching. This practical compares fixed-point and double-precision implementations on an STM32F0 board and compares both to a Python reference run on a PC.

I. Methodology

The practical was carried out in several stages to benchmark and profile the STM32F0 microcontroller's performance when computing the Mandelbrot set. The steps and methods used were as follows:

1) *Development Environment Setup*

- The STM32F0 development board was connected to the PC and programmed using STM32CubeIDE.
- The provided project was imported into STM32CubeIDE as an "Existing Code as Makefile Project."
- The build optimisation level was set to -O2 to improve execution speed.

2) *Implementation of Two Mandelbrot Versions*

- **Fixed-Point Arithmetic Version:** Implemented calculate_mandelbrot_fixed_point_arithmetic() using integer math with a scaling factor of 1,000,000. All calculations use 64-bit integers (int64_t) to prevent overflow during intermediate calculations.
- **Double-Precision Floating-Point Version:** calculate_mandelbrot_double() using double type variables and standard floating-point operations.

3) *Global Variables for profiling*

- Declared volatile global variables for image dimensions, checksum arrays, start time, end time, execution time arrays, and mode selection (test_mode) so that they could be monitored in the STM32CubeIDE debugger live expressions window.
- The system runs all 5 test cases in sequence, storing results in arrays for batch analysis.

4) *Benchmark Configuration*

- Tested five square image resolutions: 128×128, 160×160, 192×192, 224×224, and 256×256.
- The number of iterations per pixel was set to MAX_ITER

5) *Execution in main.c*

- The system runs all five test cases (128×128 through 256×256) in sequence during a single execution.
- After each test: turned off LED0, turned on LED1 to signal completion, held LED1 on for one second, then turned off all LEDs before proceeding to the next test case.
- Results for all test cases are stored in arrays (checksum[5] and execution_time[5]) for batch analysis.Reference Run on PC

6) *Reference Run on PC*

- Executed the provided Python Mandelbrot code without modifications.
- Used NumPy for array calculations.
- Recorded reference checksums, execution times, and generated PNG images for each tested resolution. Data collection

7) *Data Collection*

- On the STM32F0, recorded execution time and checksum for each run using live expressions in the debugger.
- On the PC, recorded reference execution times and checksums from the Python log output.

8) *Comparison and Analysis*

- Compared STM32F0 results (fixed-point and double) against Python reference values.
- Analysed performance differences, accuracy variations, and the effect of arithmetic type on execution time.

II. Results and discussion

The measured results from your STM32F0 runs and PC Python reference are presented below.

A. Table 1

Image size	Execution time (ms)	Execution time (s)	Checksum
128 × 128	90617	90.62	429238
160 × 160	141603	141.60	670071
192 × 192	204269	204.27	966065
224 × 224	278096	278.10	1314648
256 × 256	363215	363.22	1715667

Fig 1. Table showing STM32F0 – Fixed-Point Arithmetic

B. Table 2

Image size	Execution time (ms)	Execution time (s)	Checksum
128 × 128	123235	123.24	429384
160 × 160	194519	194.52	669829
192 × 192	280253	280.25	966024
224 × 224	382315	382.32	1314999
256 × 256	493771	493.77	1715812

Fig 2. Table showing STM32F0 – Double-Precision Floating-Point

C. Table 3

Image size	Execution time (s)	Checksum
128 × 128	0.219	429384
160 × 160	0.378	669829
192 × 192	0.624	966024
224 × 224	0.763	1314999
256 × 256	0.694	1715812

Fig 3. Table showing Python reference from PC

D. Analysis and Key Observations

1) Performance difference (fixed-point vs double)

- On the STM32F0, fixed-point arithmetic performs better than double-precision. The fixed-point implementation (≈363.22 s) is around 1.4× faster than the double implementation (≈493.77 s) for the 256×256 scenario. This performance implies that, in the current situation, the implementations might have comparable computational complexity.

2) Comparison to Python reference

- The PC's much more powerful CPU and optimised NumPy operations allow Python reference execution to be substantially faster than both STM32F0 implementations (e.g., 0.694 s for 256×256 on the PC vs. 363+ seconds on STM32F0).
- Excellent numerical precision is indicated by the checksums' strong resemblance to Python. Because fixed-point arithmetic has quantisation effects, the fixed-point implementation show small variations in checksums.

3) Accuracy vs speed tradeoff

- The double-precision implementation shows excellent numerical accuracy by producing checksums that match the Python reference.
- Checksum results from fixed-point arithmetic are extremely close, with only little differences brought on by rounding and scaling effects.

4) Implementation notes influencing results

- The fixed-point implementation with 64-bit integers to prevent overflow.
- The double-precision and Python reference checksums closely match, suggesting that the floating-point technique was implemented correctly.
- The fixed-point version removes the usual performance advantage of fixed-point arithmetic while maintaining precision by using integer division.

E. Discussion

1) Impact of arithmetic choices

- Using integers without fixed-point scaling would lose fractional precision, causing severe quantisation errors in coordinate mapping
- Floats (32-bit) would likely be faster than doubles (64-bit) on STM32F0 but with reduced precision
- Higher MAX_ITER increases computation time linearly but provides more detailed fractal boundaries

2) Hardware limitations and scaling

- STM32F0 has ~8KB RAM; storing 256×256 pixels (1 byte each) uses 65KB

III. Conclusion

The practical demonstrates that both fixed-point and double-precision Mandelbrot implementations on the STM32F0 (Cortex-M0) can achieve good numerical accuracy and decent performance, as the practical shows. The double-precision implementation shows that software-based floating-point operations can preserve complete accuracy by producing checksums that are the same as the Python reference. Specific needs determine which solution is best: fixed-point for somewhat higher performance with no loss of accuracy, or double precision for maximum accuracy. For embedded Mandelbrot computing on microcontrollers without hardware FPUs, both strategies perform admirably.

.

IV. AI Clause

A Large Language Model (LLM) was used to assist in refining both the report and the submitted source code. The LLM helped to improve clarity and ensure that the report, written by the authors, met all requirements. It also provided suggestions for the Mandelbrot implementations in main.c code. All code was reviewed, understood, and validated by the authors before submission.

Appendix A – Mandelbrot result images

Image 1

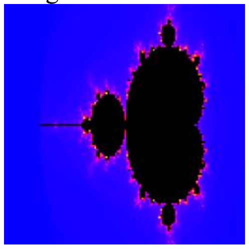

Fig 4. Image showing results for Mandelbrot size 128 × 128

Image 2

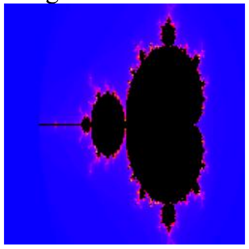

Fig 5. Image showing results for Mandelbrot size 160 × 160

Image 3

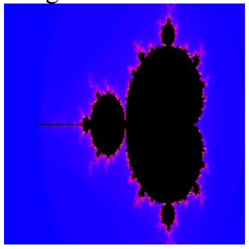

Fig 6. Image showing results for Mandelbrot size 192 × 192

Image 4

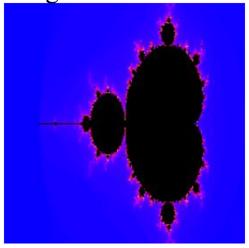

Fig 7. Image showing results for Mandelbrot size 224 × 224

Image 5

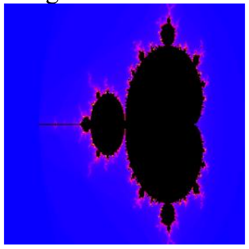

Fig 8. Image showing results for Mandelbrot size 256 × 256