CS-473: System Programming for SOCs

Practical Work 1:
The Mandelbrot set

Report

Andrea Grillo

371099

Index

Fixed-point version

Choice of the format

The first thing to choose for defining the format is the total number of bits that the data type will use. A 64-bit format would allow for better precision and bigger numbers to be stored, but the system has a 32-bit bus and registers. Using 64-bit variables would add a big overhead to the operations and slow down the computation. So the final choice is for a 32-bit variable.

The next choice is about the division of the integer and fractional parts. The choice is a trade-off between the ability to store a larger number and the precision achieved. I tried to understand what is the biggest value to be stored, to use the minimum number of bits for the integer part, including a safety margin to avoid overflow.

For the Mandelbrot set computation, the biggest values to be stored will be the sum of the variables xx and yy. As soon as this value exceeds *4.0*, the loop will be stopped, so the maximum value should not be so far away from *4.0*.

To understand what is the real maximum value that will be needed, the original algorithm has been modified to save the maximum value computed in a variable. The result is *34.924877*. As the representation will be signed, at least 7 bits are needed to store the integer part of the number.

For the final choice, another bit was added as a safety margin, so the representation chosen is ***Q8.24***, using *8* bits for the integer part and *24* for the fractional one, allowing to have *2^-24 = 0.00000006* as the quantization error.

Implementation

Below the explanation of the modifications made to the code:

- *fractal_fxpt.h:* in the header file, the *my_fixed* typedef has been defined, together with the two macros *convertToMyFixed* and *multiplyFixed*. The first one is used to convert normal integer or float values to the Q8.24 format. The other one is used to multiply two numbers in the fixed point representation. The prototypes of the *draw_fractal* and *calc_mandelbrot_point_soft* have been modified to use the *my_fixed* type instead of float, as well as the *calc_frac_point_p* pointer to function.
- *fractal_fxpt.c:* this is the file where most modifications were done. The definitions of the functions *calc_mandelbrot_point_soft* and *draw_fractal* have been adapted to the new *my_fixed* type. The *draw_fractal* function does not need any more modifications, as the only operations that are done are sums that do not need to be changed. In the *calc_mandelbrot_point_soft* function, all the multiplications have been replaced with the macro *multiplyFixed*. There is also a trick to improve efficiency of the code, which is the fast multiplication by 2 by shifting left in the computation of the *two_xy* variable.
- *main_fxpt.c:* in the main file, the modifications are just the definitions of the constants *FRAC_WIDTH, CX_0, CY_0* and *delta*. For the *delta* variable, the division is made in floating point before converting to fixed point, because I have not implemented division using fixed point.

Results

The implementation of fixed point computations leads to a major improvement of the speed of execution of the whole algorithm. The execution time of this version has been measured and is **18 seconds**.

16-bit version

During the implementation of the fixed-point version of the algorithm, at the beginning I encountered some problems multiplying 32-bits integers. This has led me to write a first version of the code using a smaller representation in 16 bits. This version is also included in the zip file and the modifications are analogous to the 32-bit version.

Having 16 bits values degrades the precision, as there are less bits for the fractional part, but this lack of precision cannot be noticed looking at the picture that is generated.

Instead, this version achieved an even better execution time of **5 seconds,** which is **more than 3x** faster than the 32-bit version. My hypothesis for this speed-up is that doing the calculations in 16-bits does not require variables bigger than the registers and bus of the system, so the multiply operation can be done easier in less CPU cycles.

Floating point version

Choice of the format

The format chosen is the same as the *IEEE-754* single-precision, that is:

- 1 bit for the sign: *0* positive, *1* negative
- 8 bits for the exponent, represented in *Excess-127*
- the remaining 23 bits for the mantissa, stored in a fixed point representation, where the integer part is 1 and just the fractional part is stored

This representation has been chosen because it allows easy conversion from float variables to the custom format, as well as easy checking of the correctness of operations done.

Implementation

For the implementation of the floating point operations, both an header file and a c file have been added:

- *myFloat.h*: contains the *changeSign* and *multiplyByTwo* macros definition, as well as the prototypes of the implemented functions.
- *myFloat.c*: contains the implementations of the functions *sumMyFloat*, *multMyFloat* and *lessThan*. A more thorough explanation of the functions is provided in the comments of the code.

Below the explanation of the modifications made to the existing files**:**

- *fractal_fxpt.h:* in the header file, the *myFloat.h* file has been included and the prototypes of the *draw_fractal* and *calc_mandelbrot_point_soft* have been modified to use the *myFloat* type instead of float, as well as the *calc_frac_point_p* pointer to function.
- *main_fxpt.c:* in the main file, the modifications are just the definitions of the constants *FRAC_WIDTH, CX_0, CY_0* and *delta*. I computed values and manually inserted them as hexadecimal code.
- *fractal_fxpt.c:* the *calc_mandelbrot_point_soft* and *draw_fractal* have been adapted to use the previously defined functions. and this is the file where most modifications were done.

Note that no checks have been done for possible overflow problems. This should not be a problem for this application, as used values are well within the supported range.

In this type of floating point representation, the zero is not included in the standard representation (*2^exponent · mantissa* cannot be zero) and should be treated as an exceptional case. This has not been done. Probably this is the reason why there is a strange behavior of the program along the center vertical line of the image, where the *x* coordinate should be zero.

Results

The code with floating point operations implemented in software runs faster than the original one, but slower than the fixed point version.

The execution time is **138 seconds**.

Conclusion

In conclusion, fixed point representations offer fast execution, but small range of values, whereas floating point operations are slower but can handle a vast range of values thanks to the exponential representation.

It is clear that the optimization of the calculation is crucial to improve the execution time of an algorithm.

However, this could be done easily in the Mandelbrot set algorithm because we know *a priori* what is the data to be processed, that is not the case for the vast majority of algorithms that process data which varies.

Below a sum-up of the results achieved:

Algorithm	**Execution time**
Original *(value from slides)*	1251 seconds
Fixed point 32-bit	18 seconds
Fixed point 16-bit	5 seconds
Floating point	138 seconds