EEE3096 PRAC2

1st Kealym Bromley
BRMKEA001

2nd Stephan Cilliers
CLLSTE009

I. Introduction

The objective of this practical was to benchmark different bit-widths, compiler flags and thread counts to find what combination of these three parameters would yield the best percentage speed up measured against a baseline for a standard program written in C.

This practical lead to 5 core findings. The first being that the C heterodyning code ran more than 72 times faster than equivalent Python code. Using more threads does not increase speedup percentage when using a processor with fewer cores than threads. Using a bit-width that is different to the word size of the operating system yields slower execution times. Compiler flags have a noticeable effect on the speedup, although more aggressive optimisation flags are less effective on more simple programs. Finally we found that, for our specific hardware, using the flag *-O1* and a *32-bit float* on our platform yielded the best increase in performance, leading to a speedup of 45%.

II. Methodology

The time taken with no modifications to the original code was recorded by writing a bash script which executes the Python and compiled C program with no optimisations and calculates the average time using the *awk* command line. module.

A. Multi-threading

A threaded version of the *CHeterodyning.c* program, *CHeterodyning_threaded.c* which implements parallelization techniques makes use of a *Thread_Count* variable in its header file to determine how many threads it should use when running. This is commented out and defined in the compiler flags of the program to allow for easy automation i.e. *gcc -DThread_Count=2*. A bash script is used to compile the code with a different *Thread_Count* ranging from 2 to 32, doubling with each iteration and run 20 times, outputting the average time. The average time for each thread count is recorded.

B. Bit-Widths

The data type that stores the sample points is varied between *float*, *double* and *__fp16*. For each bit width, variables defined with type of float in the *globals.h* and *CHeterodyning.c* source files are replaced with the type of interest i.e. double and __fp16. A bash script is written to run the compiled code 20 times and output the average, which is recorded for each bit width test. In the case of *__fp16* the code has to be compiled with the *-mfp16-format=ieee* flag.

C. Compiler flags

Various compiler flags can be used for optimisation at compile time. The compile flags to be tested are: *-O0*, *-O1*, *-O2*, *-O3*, *-Ofast*, *-Os*, *-Og*, and *-funroll-loops*. The *CFLAGS* variable in *makefile* is removed in favour of defining it inline when running the *make* command in a bash script. A bash script is written to compile the code with a new *CFLAGS* variable for each flag to be tested and to then run the compiled code 20 times and output the average. This average time is recorded for each compiler flag.

D. Data Processing

The recorded data is compared to the baseline times recorded at the start of the experiment by means of a speedup percentage calculated with Eq. (1)

$$\text{Speedup } \% = (t_{\text{baseline}}/t_{\text{avg}} - 1) \times 100 \tag{1}$$

III. Results

A. Benchmark

As a benchmark the hetrodying code was run on both Python and C. Python yielded an execution time that was more than 72 times longer than the equivalent C code. The Python and C execution time can be seen in Table 1 below. The execution time measured while running the equivalent C code will be used to calculate the percent speedup in the coming sections.

TABLE I
Base line test

Test 0	Python (ms)	C (ms)
	549	7.59

B. Multi-threading

Using 32-bit floats and no optimisation flags, a thread count of 2, 4, 8, 16 and 32 were tested. The results of the test were added to Table 2 below.

TABLE II
Multi-threading Test

Test 1	C (ms)
2 threads	13.94
4 threads	14.74
8 threads	16.17
16 threads	18.45
32 threads	20.30

As can be seen in Fig. 1 the percentage speedup is inverse to the number of threads. This can be explained as the *Raspberry Pi Zero* has a single core processor and using multiple threads adds execution time overhead. Normally multiple threads would be distributed across the CPU cores and this overhead would to mitigated as the program would now be parallelized, potentially leading to a faster execution time compared to the added overhead. As the *Raspberry Pi Zero* is single core, there is more overhead with each increase in thread count, but no way of mitigating that as there are no more cores to distribute the processing onto.

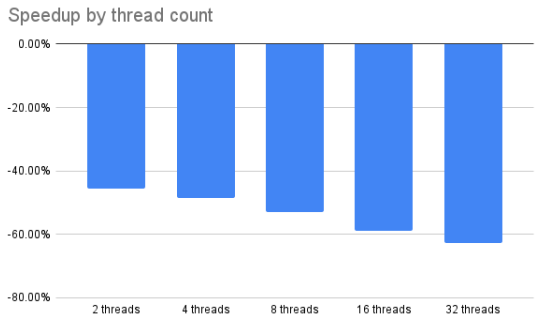

Fig. 1. Percentage speed up by number of threads

C. Bit-Widths

Three different bit-widths were tested, 64, 32 and 16 bit. Below, in Table 3, are the results.

TABLE III
BIT-WIDTH TEST

Test 2	C (ms)
64 bit (Double)	11.24
32 bit (Float)	7.56
16 bit (fp16)	29.93

It can more easily be see in Fig. 2 below that a lower bit-width does not automatically mean less computation time. The fact that the Raspberry pi zero utilises a 32-bit operating system could be the reason why the 16 and 64 bit data-types did not work as efficiently. As the data is still worked on 32 bits at a time.

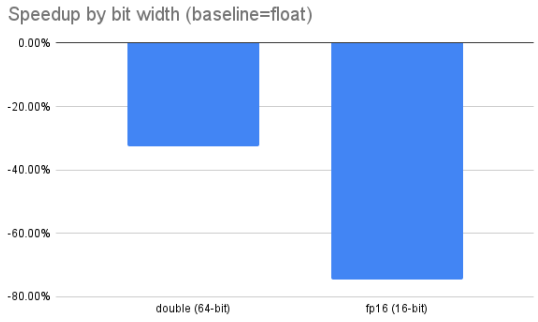

Fig. 2. Percentage speedup by bit width

D. Compiler flags

After testing the 8 different compiler flags we found that using *-O1* yielded the best results at a speed up of nearly 45%. Table 4 and a graph of the percentage speed up in Fig. 3 are below.

TABLE IV
COMPILER FLAG TEST

Test 3	C (ms)
-O0	7.470
-O1	5.216
-O2	5.388
-O3	5.383
-Os	5.386
-Og	6.818
-funroll-loops	7.433
-Ofast	5.419

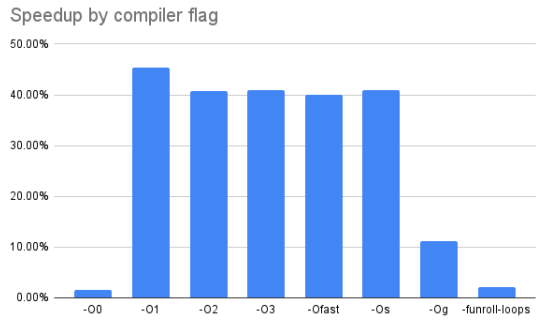

Fig. 3. Percentage speedup by bit width

As can be seen above many compiler flags yielded speedup percentages that were similar. The *-O1* compiler flag performs the least aggressive optimisations and is most likely the reason it was the fastest, as the program is simple. The program is too simple for more aggressive optimisations to have an effect on the speedup percentage.

IV. CONCLUSION

C programs can be optimised to increase performance by means of compiler flags, data types used in code, and potentially parallelization (depending on CPU architecture). Adjusting compiler flags provided at most a 45% increase in the case of *-O1*, and performed at baseline speeds in the worst case. 32-bit data storage mechanisms i.e. *float* was found to perform better than *double* and *__fp16* on the Pi due to the 32-bit CPU architecture. Parallelization was not effective on the Pi because it only has one CPU core and hence parallelization only introduced overhead, which slowed the program down the more threads were used.