Optimisation of Code on a Raspberry Pi

Jonathan Campbell[†] and Akhil Jacob[‡]
EEE3096F Class of 2021
University of Cape Town
South Africa
[†]CMPJON005 [‡]JCBAKH001

***Abstract*—In this practical focus on how fast different programming languages on a Raspberry Pi operate. It also looks at the optimise of the code to make it work faster on a Raspberry Pi.**

I. Introduction

Optimization is the process of making the best use of the available resources. In the embedded environment to complete a task there will not always have the necessary resources need to run the task. This could be the clock frequency of a micro-controller only reaches to 5MHz or the micro-controller can only work with 16 bits all while competing task on time and be energy-efficient. a good example is the Apollo guidance computer [1].

The platform the was used is the Raspberry Pi. This was programmed to compute a series of tasks to complete and return the time it took to complete those tasks. This uses a ARM11 CPU with a clock speed of 1GHz and has 512 MB RAM, with a SoC of Broadcom BCM2835 [2].

It is know that python is a very slow programming language because it is a interpreted language and not a compiled language like c or c++ [3]. Thus for this practical Python will be used as a baseline measurement and optimizing the c code by changing the multi-threading, Bit widths and compiler flags.

II. Methodology

The effect of changing the thread count, flags and Bit width of C code.

A. Hardware and Software

For the practical all the software that was used was taken form GitHub. The Python and C programming languages the compilers where already on the Pi as they are standard in the Pi OS. The Hardware used was one Raspberry Pi Zero with a heat sink and was connected to a 5V 2A power supply via a micro-USB cable. The full code that was used is below in the Appendix. Code snippets are shown below.

B. Implementation of Code

Python source Code

```python
# import Relevant Librares
import Timing
from data import carrier, data

# Define values.
c = carrier
d = data
result = []

# Main function
def main():
    print("There are {} samples".format(len(c)))
    print("using type {}".format(type(data[0])))
    Timing.startlog()
    for i in range(len(c)):
        result.append(c[i] * d[i])
    Timing.endlog()

# Only run the functions if this module is run
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("Exiting gracefully")
    except Exception as e:
        print("Error: {}".format(e))
```

The line that was changed to make the CHeterodyning_threaded.c run with multiple threads

```
//------------------------------------------------------------------------------
#define Thread_Count 1
//------------------------------------------------------------------------------
```

Implementation of changes to the C code
The optimisation of the makefile

```
CFLAGS = -lm -lrt -funroll-loops
```

The changes to the globals.h file

```
#define SAMPLE_COUNT 100000
float carrier[SAMPLE_COUNT] = {0.0, 0.963507348203431, ...}
float data[SAMPLE_COUNT] = {0.0, 0.100488636359539, ...}

#define SAMPLE_COUNT 100000
double carrier[SAMPLE_COUNT] = {0.0, 0.963507348203431, ...}
double data[SAMPLE_COUNT] = {0.0, 0.100488636359539, ...}
```

Script for running the code automatically

```
for i in (1..10); do make run_threaded >> c_32.txt; done
nano c_32.txt
```

C. Experiment Procedure

The first task that was done was to run the Python code on the Pi to establish a ‘golden measure’. Then the unoptimized C code that was provided on the git repo was compiled and ran 10 times to obtain an average with the last 5 results from both tests where recorded.

Next, the number of threads in CHeterodyning_threaded.h was changed and compiled then ran this was repeated for 2, 4, 8, 16 and 32 threads. Each thread was run 10 times and the last 5 where chosen and an average value was taken to improve accuracy this was done as the CPU takes longer in

the beginning as it is not use to the code. The next part required the change of the flags to the makefile and run it 10 times with the last 5 being recorded and the average speed was taken to see how the added flag helps optimize the code. This process took very long as it was initially done manually.

Seeing that the method of running the code manually many times and recording the values was time consuming and errors could be made, a script was implemented to run and record the values of the speeds into a text file which was then used to calculate the average value. The script can be found in the Appendex This sped up the process and showed us exactly how important automation is in the experiment.

Lastly,the code was optimized by using different bit-widths this was done by changing the code in the globals.h file to double, float, and fp16. The results where recorded from the different bit-widths, float, double and __fp16. An extra flag, -mfp16-format=ieee, was needed in the makefile to run it.

III. Results

A. Baseline tests

These results where from ruining the Python and C programs multiple times to get the average. In Table 1 below it can be noted that Python code runs signify slower at about 50 times slower than the C code. The threaded C program runs at 2 times slower than the C program but it runs 28 times faster than the Python program.

TABLE I: Baseline test Data

	Python	**C**	**C threaded**
1	583,20	9,75	20,53
2	559,32	8,05	21,45
3	599,36	14,84	21,36
4	537,74	8,09	20,38
5	587,47	8,95	20,48
Average	573,42	9,94	20,84

B. Threaded Test

In Table 2 below shows the results form the threaded test. This is when the amount of threads where changed which makes the CPU run only that a single instruction at a time with multiple threads the CPU is allowed to run multiple instructions the same time. This data shows that 8 threads is the best as it has the lowest time.

TABLE II: Threaded Test Data

	1thread	**2thread**	**4thread**	**8thread**	**16thread**	**32thread**
1	7,25	11,74	9,71	12,04	9,67	10,20
2	10,4	7,23	13,94	7,30	11,14	11,76
3	10,42	12,05	9,36	11,01	8,61	11,66
4	11,24	10,24	8,11	9,82	11,96	10,18
5	11,43	10,19	14,57	9,34	8,76	9,79
Avg	10,14	10,29	11,13	9,90	10,02	10,71

C. Flags Test

In Table 3 it shows 6 flags being run. The results from this test show that -O2 flag works the best. As this forces on speed and not on size or compiler speed. The slows is the -O0 flag which does not do any optimisation and just makes debugging easier.

TABLE III: Flag Test Data

	O0	O1	O2	O3	Ofast	Os	Og	funroll-loops
1	12,97	9,99	5,21	6,43	5,36	6,29	9,25	8,34
2	9,84	7,01	7,11	7,28	7,12	7,97	9,21	8,61
3	7,38	7,97	5,25	8,36	5,26	10,15	9,24	13,61
4	11,32	6,91	7,15	10,73	8,36	7,03	8,02	9,9
5	8,94	7,05	6,46	6,46	8,82	7,31	9,23	13,02
Average	10,09	7,79	6,24	7,85	6,98	7,75	8,99	10,70

D. Bit Widths

A bit width controlled how many bits are used to store a number in memory. A float default bit width is usually 32 bits while double is normally 64 bits. In the table below it shows that the float is the faster than double as the the CPU does not have to read and write to 2 times more bits in memory which takes time. What is surprising is that __fp16 is not the fastest as it only uses 16 bits thus less than float.

TABLE IV: Bit Width test Data

	Float	double	__fp16
1	12,27	17,01	33,36
2	9,12	12,67	39,56
3	10,84	15,45	37,52
4	12,31	14,56	45,67
5	11,51	17,62	44,7
avergae	11,21	15,46	40,16

IV. Conclusion

In comparing the different tests it is found that the fastest time is when the C code is run with 8 threads with the -O2 flag and having float bit width. This can be understood as the more threads a program runs on the faster it is but if there are to many threads it can slow done the program as only one thread can read and write memory at a single time [3]. With regards to the float being faster than __fp16 , what is probably causing this is that the C program could be comparing data and there are not enough bits it get an accurate reading thus it is running the program to get a more accurate reading, more research need to be done on this.

More research is necessary with different C programs to test the reliability of the optimisation as we have only used optimised one file. Also we only optimised for the speed of the program in a further study the C program could be optimised to have the smallest memory space with the highest speed it all depends on the situation.

V. Links

https://github.com/Jonathan5320/EEE3096S.git

References

[1] J. Ganssle, "Engineering apollo," Sep 2008. [Online]. Available: https://www.embedded.com/engineering-apollo/

[2] L. Hattersley and L. is Editor of The MagPi, "Raspberry Pi 4, 3A+, Zero w - specs, benchmarks amp; thermal tests." [Online]. Available: https://magpi.raspberrypi.org/articles/raspberry-pi-specs-benchmarks

[3] O. published by Anthony Shaw on, "Why is python so slow?" [Online]. Available: https://hackernoon.com/why-is-python-so-slow-e5074b6fe55b