Emulation and Benchmarking of C Code

Kian S Frassek† and Cameron F Clark‡
EEE3096S Class of 2023
University of Cape Town
South Africa
†FRSKIA001 ‡CLRCAM007

***Abstract*—This report details the emulation and benchmarking of C code with various compiler optimisation. Python was using as our Golden Measure, and C code with 8 threads, floats, -Os and unrolled loops being the fastest**

I. Introduction

This report covers an investigation into the topics of emulation, benchmarking, and optimisation in embedded systems development. The aim is to uncover the impact of the chosen high-level language, compiler flags, bit widths, and multi-threading on a program's performance.

Emulation is a useful tool that allows the simulation of hardware and software simultaneously, and benchmarking is used to give a reference frame for all the tests. Python is chosen as our Golden Measure for these tests for its simplicity while the C code is optimised to increase the speed of the program.

Optimizations are tools used during the compiling process that are designed to improve program performance in one or more key aspects. These optimizations include changing the number of bits that the variables are stored in, adding various optimisation flags and the use of multi-threading.

II. Methodology

A. Hardware

The hardware used for this practical was a macOS laptop running Virtual Box with 4 CPU cores and 8 Gb of RAM. The Virtual Box was running Ubuntu image, which was in turn simulating a Raspberry Pi emulator. The Raspberry Pi emulator was used for the tests.

B. Implementation

The following code is the sudo code format of the two files: the python code which was used as the benchmark for the C code tests and the non-threaded C code. The third file that was used in this test was a multi-threaded version of the C code file. The 'data' and 'carrier' arrays are predefined arrays found in another tile

```
# Load data arrays from external file
Load data from external file

# Define values
c = carrier array
d = data array
result = empty array

# Main function
function main():
 Display "There are " + length of c + " samples"
 Display "using type " + type of first element in data array
 Call Timing.startlog()
 for i from 0 to length of c - 1:
 Append c[i] * d[i] to result array
 Call Timing.endlog()
```

C. Experiment Procedure

The experiment start by benchmarking in Python using the following code. All tests were run 5 times and then an average was taken:

```
cd Python
for i in {1..5}; do python3 PythonHeterodyning.py; done
```

The C code was tested next starting with the non-threaded version:

```
cd ../C
make
for i in {1..5}; do make run; done
```

The threaded version of the C code was tested next, where each test changed the defined number of threads in the *CHeterodyning_threaded.h* file. The nano command was using to edit the file:

```
nano src/CHeterodyning_threaded.h
for i in {1..5}; do make run; done
```

Where the thread count was changed in the following line to the values 1, 2, 4, 8, 16, and 32 in turn:

```
#define Thread_Count 1
```

The threaded C code was then executed for each change of thread count:

```
make threaded
for i in {1..5}; do make run_threaded; done
```

The next test involved the use of compiler optimisation flags which optimisation which optimise the code for various application. One of the following were using at a time:

- -O0
- -O1
- -O2
- -O3
- -Ofast
- -Os
- -Og

They were added to the The *makefile* under the *$(CFLAG)* heading and then the non-threaded version was run:

```
nano makefile
```

The *-funroll-loops* flag unrolls loops into repetitive code. It can be combined with any of the flags used in the previous test, however, for efficiencies sake, it was only applied to the slowest and fasted two compiler optimisation flags.

The last test involves altering bit widths of the data arrays. This involves change the data type of the arrays in *globals.h* and the output array in *CHeterodyning_threaded.h* The data types that were tested were

- float
- double
- __fp16

The following code quickly changes the data types in the array where *abc* and *xyc* can be substituted as to the data type in the file and the data type to change to respectively

```
sed -i -e 's/abc/xyz/g' src/globals.h
sed -i -e 's/abc/xyz/g' src/CHeterodyning.c
```

Importantly, when the data type *__fp16* is used, the compiler flag *-mfp-format=ieee*. All added compiler flags were removed before running this test, and the non-threaded version was used

Finally the best combination of the above number of threads, compiler flags including *-funroll-loops* and bit width was used to generate the fastest C code execution.

III. Results

This section will include details about all the results gathered throughout this experiment in a table format.ThiS allowing for comparisons between the different variations of Compling a C program.

A. Tables

B. Figures

Include good quality graphs. These were produced by the Octave code presented in listings **??** and **??**. You can play around with the `PaperSize` and `PaperPosition` variables to change the aspect ratio. An easy way to obtain more space on a paper is to use wide, flat figures, such as Fig.

TABLE I

Benchmarking and Non-threaded tests and Times (ms)

	Python	**C**
Test	Golden Measure	Non-threaded
1	403	11,1
2	412	15,1
3	406	6,12
4	367	7,07
5	383	7,65
Average	394	9,41

TABLE II

Threaded tests and Times (ms)

	Threads					
Test	1	2	4	8	16	32
1	11,2	1,59	17,3	8,36	1,46	3,40
2	10,5	11,2	6,71	11,3	8,90	4,88
3	25,5	31,2	22,1	10,9	26,0	24,4
4	40,4	37,5	2,59	11,3	2,07	17,8
5	14,9	52,0	7,66	1,65	24,8	29,5
Average	20,5	26,7	11,3	8,70	12,6	16,0

TABLE III

Compiler Optimisation Flags tests and Times (ms)

	Optimisation Flags						
Test	-O0	-O1	-O2	-O3	-Ofast	-Os	-Og
1	30,4	14,1	28,2	28,7	7,10	21,0	32,8
2	34,7	33,3	20,9	26,9	12,3	5,78	8,92
3	8,45	8,09	36,6	6,02	19,0	5,03	21,8
4	12,1	6,19	5,31	9,71	5,87	24,2	6,22
5	35,1	48,4	7,77	53,4	29,1	5,73	25,3
Average	24,2	22,0	19,8	24,9	14,7	12,3	19,0

TABLE IV

COFs with *-funroll-loops* tests and Times (ms)

	Optimisation Flags			
Test	-O0	-O3	-Ofast	-Os
1	43,9	7,31	24,6	6,01
2	6,18	6,78	5,08	5,67
3	12,1	15,1	42,7	11,8
4	31,7	57,9	6,06	21,1
5	13,4	34,0	6,21	5,47
Average	21,5	24,2	16,9	10,0

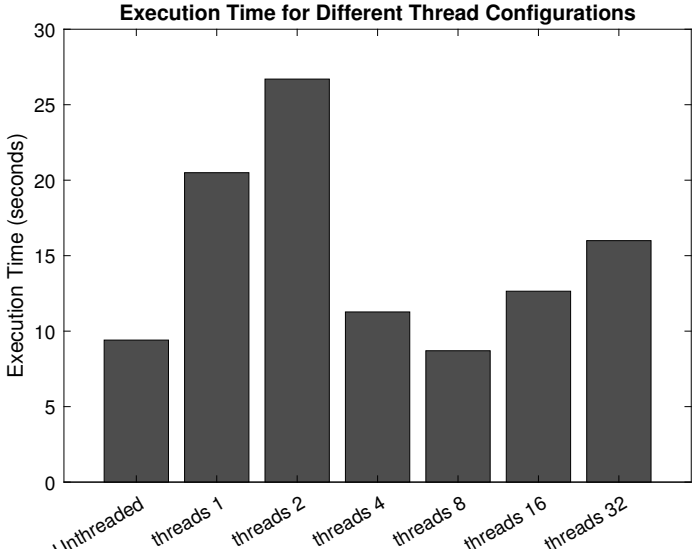

TABLE V

C BIT WIDTHS TESTS AND TIMES (MS)

	Data type		
Test	double	float	__fp16
1	31,9	6,69	110
2	39,2	31,1	101
3	34,1	10,9	26,2
4	69	35	98,8
5	12,1	7,87	93,8
Bit widths	64	32	16
Average	37,26	18,312	85,96

TABLE VI

FASTEST TEST

Section	Fastest	Test	Time (ms)
Language	C	**1**	14,3
Threaded	True	**2**	1,68
Threads	8	**3**	14,6
Compiler Flag	-Os	**4**	1,90
Unroll loops	True	**5**	21,7
Data type	float	**Average**	10,8

IV. CONCLUSION

The conclusion should provide a summary of your findings. Many people only read the introduction and conclusion of a paper. They sometimes scan the tables and figures. If the conclusion hints at interesting findings, only then will they bother to read the whole paper.

You can also include work that you intend to do in future, such as ideas for further improvements, or to make the solution more accessible to the general user-base, etc.

Publishers often charge "overlength article charges" [1], so keep within the page limit. In EEE4084F we will simulate overlength fees by means of a mark reduction at 10% per page. Late submissions will be charged at 10% per day, or part thereof.

REFERENCES

[1] "Voluntary Page and Overlength Article Charges," http://www.ieee.org/advertisement/2012vpcopc.pdf, 2014.