Practical 2 - Python vs. C

Ronak Mehta[†] and Vikyle Naidoo[‡]
EEE3096S Class of 2019
University of Cape Town
South Africa
[†]MHTRON001 [‡]NDXVIK005

***Abstract*—Performance and optimization of the C code in comparison to the python code using a combination of parallelization, compiler flags, bit widths, and hardware level features**

I. Introduction

This practical investigated and compared the performance between C and python code in the context of embedded systems development. It started by running a program in Python, and comparing it to the same program but written in C. Thereafter, it optimized the C code, by using different bit widths, compiler flags, particular hardware available in the ARM Processor, as well as implementing threading to improve performance. The quest for optimization could be a long, unending spiral but through benchmarking the results, this report tried to find a suitable performance comparison between the two languages which proved to be vital in deducing the importance of using C over Python in terms of speed up ratio.

II. Methodology

This section will outline the hardware used and describe the different methods utilized in optimizing the C code

A. Hardware

This experiment required the use of a Raspberry Pi 3B+, an Ethernet cable and a SD card. The Pi was used as a tool to carry out the experiment, the Ethernet cable served as means of internet connectivity and the SD card is where our Raspbian image was stored

```
Architecture:        armv7l
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               4
Model name:          Cortex-A53
Stepping:            r0p4
CPU max MHz:         1400.0000
CPU min MHz:         600.0000
BogoMIPS:            38.40
Flags:               half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
```

Fig. 1: Computer Hardware

B. Establishing the Golden measure

The golden measure was established by running the Python code which was obtained under Prac2 source files from github and taking an average of three runs, with the first run being discarded (to clean the cache).

C. C code

To compare between C and Python, the C code was run without optimizing and the average run time was recorded. The C code was run using floats similar to the Python implementation and the execution of this code was done by extracting it from Prac2 source files and using the **make** command to compile the Prac2.c file. It then used the **make run** command to run Prac2. With the first run being discarded (for cache cleaning), an average of four runs were recorded to obtain the run time for the C code without optimization.

D. Optimizing the C code

This subsection will now describe and discuss the different methods used for optimizing the C code. The methods used in this experiment are as follows:

1. Optimizing through multi-threading
2. Optimizing through compiler flags
3. Optimizing through bit-widths
4. Optimizing using hardware level features
5. Optimizing using a combination of the above methods

1) Optimizing through multi-threading:
C is a language which by default operates on a single thread (i.e. it only runs the code one instruction at a time). Utilizing multi-threading in a code is analogous to multitasking, in the sense that it allows for two or more parts of the code to run simultaneous. To achieve multi-threading, a threaded version of Prac2 was compiled using the **make threaded** command and run using **make run_threaded**. This experiment ran the C code for 2 threads, 4 threads, 8 threads, 16 threads and 32 threads. The number of threads were defined in the Prac2_threaded.h file under the Prac2 folder in github. This experiment also made use of the **make clean** command to clean the binary and object files that had been compiled already, to run a new set of instructions. Finally, an average of four runs were recorded for each number of threaded version in order to compare which one fits best for optimization.

2) Optimizing through compiler flags:
Optimization through compiler flags was performed on the non-threaded C code. The makefile was edited each time to include compiler flags (O0, O1, O2, O3, Ofast, Os, Og and funroll-loops). An average of four runs were recorded for each compiler flag in order to compare which one fits best for optimization. For the purpose of this experiment, the flag funroll-loops was used only in conjunction with O0 and Ofast.

3) Optimizing through bit-widths:
The next method of optimization was done through the modification of bitwidths. Here three data types were compared:

- Float: A float is 32 bits with a precision size of 4 (This is the default used in the previous experiments)
- Double: This is 64 bits with a precision size of 8.
- __fp16: This is 16 bits with a precision size of 2. From 1, we see that __fp16 is not an arithmetic data type. Its values automatically promote to single-precision float or double-precision double floating-point data type when used in arithmetic operations, and are then converted to the half-precision __fp16 data type for storage. NB: There is a need to specify **-mfp16-format=ieee** under CC flags in the makefile for __fp16.

These bit-widths of data were then changed in the globals.h file and the calculations were done using the respective datatype in Prac2.c as well. The run times due to each bitwidth were recorded, and the averages of each were then calculated.

4) Optimizing using hardware level features:
The Pi has various instruction sets. This experiment sets the floating point hardware accelerator with the **-mfpu=**(any HW level feature) command. The hardware level features used for this experiment were vfpv3, vfpv4, neon-fparmv8 and vfpv3xd. These features were individually applied (by editing the makefile), and their respective runtimes were recorded for comparison.

5) Optimizing using a combination of the above methods:
Finally, a combination of bit-width, compiler flags and hardware level features were used together in order to find the best possible optimization and speed up over the golden measure implementation.

III. Results

This section will present and discuss the findings from the experiment conducted both with and without optimizations

A. Without Optimization

Both the Python code and C code were run and an average of 3 runs were taken for each to get a value for comparing with each other:
Python code average runtime = 8.880456 seconds
C code average runtime = 7.325481 milliseconds
Speed up = Python runtime / C runtime = 1212.3
Hence, without optimization, the C code is approximately 1200 times faster than the Python code.

B. With C code optimization

The following results obtained were recorded when the C code was optimized using different features. For all optimization methods, the average was taken of four runs, disregarding the first run as it was aimed at cache cleaning. The Speed up was calculated using the ratio of average runtime of the Unoptimized C code to the average runtime of the newly optimized code (independent of other optimization methods). The following results were obtained during the length of this practical:

TABLE I: C Code Optimization using multi-threading

	Run Time (ms)				
	2 Threads	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
Run#2	6.12363	3.75950	3.44755	2.52909	1.87431
Run#3	6.19800	3.83044	2.98595	2.97377	1.90915
Run#4	6.19676	3.62674	3.57927	3.39403	1.95816
Run#5	6.01551	4.19293	3.72911	1.43743	1.82863
Average:	6.13348	3.85240	3.43547	2.58358	1.89256
Speed up:	1.19434	1.90154	2.13231	2.83540	3.87067

Deducing from the results above, it was evident that the benchmark ran faster every time there was an increase in the number of threads. The most performant one was the one with 32 threads, having an average run time of 1.89ms and a speedup of 3.87 (387%). This outcome makes sense because having 32 executions of the code at the same time allows maximum utilization of the CPU by means of multitasking.

TABLE II: C Code Optimization using compiler flags

	Run time (ms)								
	O0	**O1**	**O2**	**O3**	**Ofast**	**Os**	**Og**	**funroll loops with flag O0**	**funroll loops with flag Ofast**
Run#2	7.213656	3.090267	2.947453	2.930006	2.990652	3.000937	5.543989	7.306788	2.218691
Run#3	7.209073	2.940426	3.102086	2.949537	3.077893	3.175419	5.660916	7.266787	2.260982
Run#4	7.213917	2.935426	2.949381	2.940110	2.994297	3.163543	5.606957	7.307620	2.502960
Run#5	7.321311	2.950529	2.967557	2.952245	2.959765	2.939997	5.613728	7.207776	2.271712
Average:	7.239489	2.979162	2.991619	2.942975	3.005652	3.069974	5.606398	7.272243	2.313586
Speed up:	1.011878	2.458906	2.448667	2.489142	2.437235	2.386170	1.306629	1.007321	3.166288

Without using funroll loops, the most performant one was O3 with an average run time of 2.94ms and a speedup of 2.49 (249%). This makes sense because the effect of the O3 flag states ”many extensive optimizations for speed” which infers that its main objective is to increase speed as opposed to size or debugging.
But the fastest speedup was obtained using funroll loops with Ofast. This combination gave a speedup of 3.17 (317%). Funrolls create loops in repeated assembly and when combined with Ofast, it improves the speed vastly but at the cost of size.

TABLE III: C Code Optimization using bitwidths

Run time (ms)			
	Float	**Double**	**__fp16**
Run#2	7.364334	8.781763	30.253049
Run#3	7.360793	8.755466	30.238055
Run#4	7.376679	8.657305	30.326702
Run#5	7.200116	8.726935	30.163907
Average:	7.325481	8.730367	30.245428
Speed up:	1.000000	0.839080	0.242201

Here, the most performant one in terms of speed was the optimization using Float. Float is 32 bits with a precision size of 4. This gave an average runtime of 7.33ms with a speedup of 1.00 (100%).In terms of accuracy, the optimization using double was the most accurate one with a precision size of 8 (a double is 64 bits)

TABLE IV: C Code Optimization using hardware level support

Run time (ms)				
	vfpv3	**fpv4**	**neon-fp-armv8**	**vfpv3xd**
Run#2	7.298121	7.154726	7.290558	7.157852
Run#3	7.410255	7.328682	7.293370	7.159779
Run#4	7.313173	7.306755	7.269048	7.221080
Run#5	7.311662	7.304725	7.336131	7.152071
Average:	7.333303	7.273722	7.297277	7.172696
Speed up:	0.998933	1.007116	1.003865	1.021301

From the results above, none of them gave a distinct difference in terms of speed and mostly all had speed ups close to 1.00 (100%). This outcome seems logical as hardware optimizations usually focus on increasing efficiency by reducing power consumption in the memory as opposed to increasing percentage speed up.

TABLE V: C Code Optimization using combination of multi-threading, bitwidths, compiler flags & HW level features

Run time (ms)				
	Float O3-funroll loops fpv4 with 32 threads	**Float O3-funroll loops fpv4**	**Float O3-funroll loops vfpv3xd**	**Float O3-funroll loops**
Run#2	1.49687	2.312059	2.490147	2.269674
Run#3	1.22094	2.255863	2.261351	2.272799
Run#4	1.23557	2.256227	2.275987	2.253216
Run#5	1.30787	2.237687	2.307861	2.389883
Average:	1.315313	2.265459	2.333837	2.296393
Speed up:	5.569384	3.233552	3.138815	3.189994

In general, combining different methods of optimization yield in better and increased speed up. The best and most performant from the lot tested in this practical was the optimization which used a combination of **32 threads** (as multi-threading optimization), **float** (as bitwidth optimization), **O3 funroll loops** (as compiler flag optimization) and **fpv4** (as hardware level support optimization). This combination gave an average run time of 1.32ms and a speed up of 5.57 (557%)

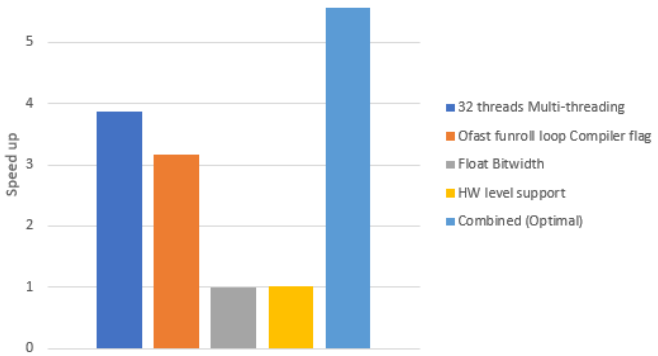

Fig. 2: Chart showing best speed up ratios from each optimization method

IV. Conclusion

This practical shows that the performance and optimization of the C code was far better than the Python code. From Section III-A, it was clear that the C code without optimization, had a speed up of nearly 1200 over the golden measure implementation. It was also clearly evident from Fig. 2 that the C code when fully optimized, ran fastest when it had a combination of **32 threads** (as multi-threading optimization), **float** (as bitwidth optimization), **O3 funroll loops** (as compiler flag optimization) and **fpv4** (as hardware level support optimization). This combination gave an average run time of 1.32ms and a speed up of 5.57 (557%) which was the fastest from all the possible methods and combinations experimented during this practical.

References

[1] Arm keil: fp16 Explanation
http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_sex1519040854421.htm

[2] Prac2 Vula Document brief
https://vula.uct.ac.za/access/content/attachment/c7c9b0c3-208f-4d61-b995-6ff0c0b9976e/Assignments/74d50871-0d0d-4f0c-9dc2-dc0102cb9fb1/EEE3096S%202019%20Prac2.pdf