EEE3096 2023 Practical 1

Md Shaihan Islam
Department of Electrical Engineering
University of Cape Town

***Abstract*—This document contains details on different methods of testing programs in order to achieve the fastest computing time.**

***Index Terms*—thread, bitwidth, float, double**

I. Introduction

Every program has a certain execution speed, which outlines the speed at which program runs successfully. It is very important practise to try to reduce the time taken to compute a certain program. In this practical, various methods will be explored in an attempt to make a simple, but computationally expensive program, faster. Execution speeds will be compared between different languages such as Python and C++, and further optimisation will be touched on when testing the C code.

Results will be recorded for all the different tests, and the execution time and speedup will be graphed to easily compare performance for each experiment. These graphs will enable the confirmation of certain performance expectations, as well as highlight interesting behaviours of patterns in the results. From analysis of the results, solid conclusions can be made.

II. Methods

A. Experiment 1 - Comparison of Python and C program runtimes

For the first experiment, the runtimes for Python and C code will be measured and compared. Both the Python and C programs will be run 5 times, and the average of the runtimes will be recorded as the final answer. To run the Python program, the command ”python3 PythonHeterodyning.py” will be called, and to run the C program, the command ”make” will be called to update the makefile with any possible changes to source files and the command ”make run” will be used to run the code.

B. Experiment 2 - Optimisation through multi-threading

The C code will then be optimised by using different numbers of threads in the code. The number of threads can be changed by navigating to the appropriate threading source file. Tests will be conducted for 2, 4, 8, 16 and 32 threads. As before, each of these tests will be conducted 5 times, and the average runtimes will be recorded.

Identify applicable funding agency here. If none, delete this.

C. Experiment 3 - Optimisation through compiler flags

The C code can be further optimised using compiler flags in the makefile. The following flags will be tested: O1, O2, O3, Ofast, Os, Og and funroll-loops. Once again, with each of the flags, the tests will be run 5 times and the average will be taken. Furthermore, different combinations of compiler flags will also be tested for further efficiency.

D. Experiment 4 - Optimisation using bit widths

In this experiment, the C code will be optimised by testing various bit-widths. The bit widths that will be tested include: fp16 (16 bit), float (32 bit) and double (32 bit). For each bit width, the average of 5 test runs will be recorded for comparison.

E. Experiment 5 - Combinations of optimisations

For the final part, different combinations of compiler flags and bitwidths will be analysed and tested to achieve maximum optimisation.

III. Results

A. Python versus C

The table below displays the output for all five tests of both C and Python code:

TABLE I
Python vs C

Test Run	Python (ms)	C (ms)
1	3,66127	6,324987
2	3,04553	45,147908
3	3,33173	15,101969
4	2,49527	15,090969
5	3,67675	44,196909
Mean	3,24211	25,1725484

B. Thread performance

The table below displays the output for all 5 tests performed for 2, 4, 8, 16 and 32 threads:

Test Run	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
1	16,1823	17,1373	13,7334	3,10009	3,39299
2	4,887	3,44294	50,9627	22,6518	8,52252
3	7,9471	15,8073	8,89159	4,45057	3,33213
4	30,615	11,5298	3,85782	12,6938	4,14823
5	17,014	8,96885	35,7373	4,95551	13,9058
Mean	15,32908	11,377238	22,636562	9,570354	6,660334

The graph on the following page depicts the speedup (vs original C code) with relation to the number of threads:

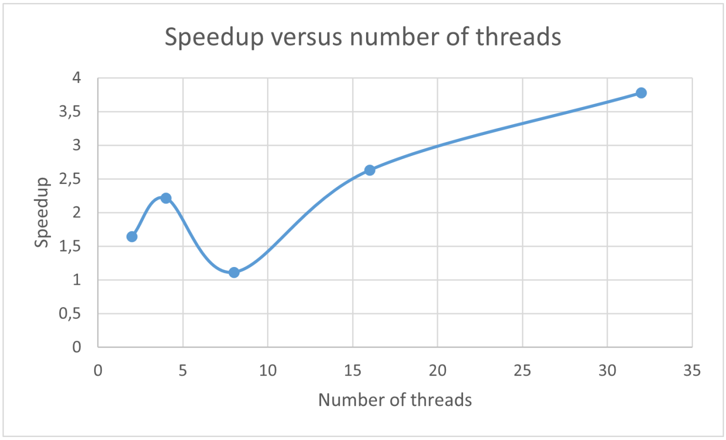

Fig. 1. Thread performance

C. Compiler flag performance

The table below displays the test results of running various compiler flags:

TABLE II
COMPILER FLAG PERFORMANCE

Test Run	O0	O1	O2	O3
1	3,39299	19,5406	3,63802	3,26307
2	16,4961	13,9461	22,9363	15,1012
3	16,4961	14,2902	32,4964	6,63808
4	4,4384	14,1121	3,27404	26,2243
5	3,24903	4,89605	2,47503	19,5642
Mean	8,814524	13,35701	12,963958	14,15817

Test Run	Ofast	Os	Og	Og funroll loops
1	2,84907	2,66107	18,1735	3,11109
2	21,8483	15,0732	3,19104	5,3539
3	9,37912	4,87906	24,7113	9,61281
4	15,5472	2,45003	37,6845	6,94686
5	10,4041	23,6223	22,8803	12,4917
Mean	12,005558	9,737132	21,328128	7,503272

The graph below visually depicts the computing time for each compiler flag:

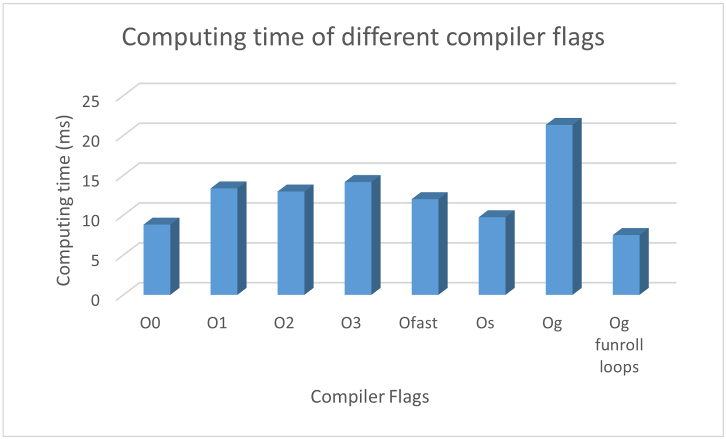

Fig. 2. Compiler Flag performance

D. Bitwidth performance

The table below displays the test results of running the program in various bitwidths:

TABLE III
BITWIDTH PERFORMANCE

Test run	float (threaded)	float (unthreaded)	double (threaded)	double (unthreaded)	fp16 (threaded)	fp16 (unthreaded)
1	3,60306	6,853331	4,573	59,20004	3,37695	100,118169
2	24,481	12,584015	7,595	38,085026	3,31595	37,537688
3	8,549	5,5440006	3,125	26,45961	4,97993	37,596686
4	3,769	5,769006	3,39	8,789006	3,24996	106,082114
5	7,058	5,648006	3,11	34,961654	7,0429	98,906172
Mean	9,492012	7,27967172	4,3586	33,4990672	4,393138	76,0481658

E. Performance of different combinations of compiler flags and bitwidths

For this section, three different combinations of compiler flags and bitwidths were tested:
- Combination A: Compiler flags Ofast and funroll-loops, float (32 bit)
- Combination B: Compiler flags O3 and Ofast, fp16 (16 bit)
- Combination C: Compiler flags O2, O3, Ofast and funroll-loops, double (64 bit)
The table below summarises the test results for the different combinations:

TABLE IV
PERFORMANCE OF VARIOUS COMBINATIONS

Test run	A	B	C
1	4,36402	9,14094	2,37902
2	5,37203	2,40098	2,36003
3	11,4101	2,56798	2,37403
4	6,27403	2,34798	5,54106
5	3,40601	4,65997	2,31502
Mean	6,165238	4,22357	2,993832
Speedup (vs original C code)	4,082981063	5,960017237	8,40813713
Speedup (vs Python code)	0,525869399	0,767623125	1,082929837

In the table above, the speedup has been compared to not only the original, non-optimised C code, but also to the ”golden measure python code. This was done by dividing the average computing time of the non-optimised code (and the Python code) by the computing time of each of the combinations.

The graph below compares the performance of each combination:

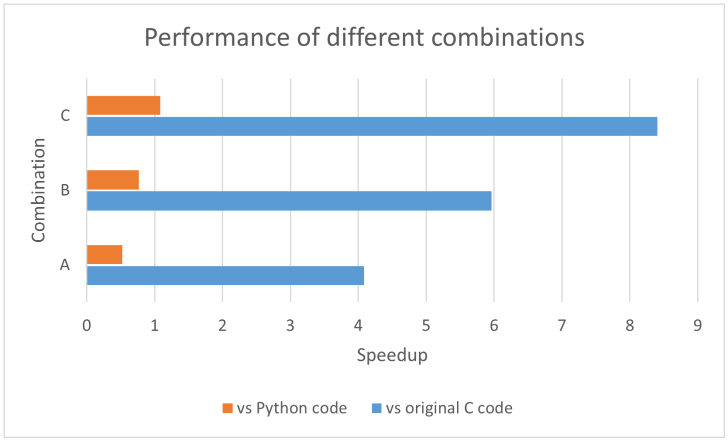

Fig. 3. Compiler Flag performance

IV. Discussion of results

A. Experiment 1 - Python vs C

This was a simple experiment in which it was very easily seen that the python code, the ”golden measure”, was much faster than the C code (7.76 times faster).

B. Experiment 2 - Multithreading

Initially the number of threads was set to 1, however, increasing the number of threads did produce a steady increase in the speedup of the program. This experiment also yielded a very interesting result, as the performance dropped significantly when 8 threads were used. There could be external factors related to this, or the parallel program may just have taken more time to compute at that particular time.

C. Experiment 3 - Compiler flags

Most compiler flags provided similar levels of optimisation, with the exception of Og. What was important to note in this experiment is the effect of the addition of the flag funroll-loops, as the addition of this simple flag turned the slowest flag into the fastest!

D. Experiment 4 - Bit-widths

Of the three bit-widths tested, float proved to be the slowest, while both fp16 and double seemed somewhat equal in performance. This experiment was run threaded (32 threads) and unthreaded, which highlighted the huge difference multithreading makes in computational time.

E. Experiment 5 - Combinations

As expected, the combination with the most compiler flags for speed optimisation, paired with the fastest recorded bitwidth, performed the best, and was also the only combination optimised enough to be faster than the ”golden measure” python program.

V. Conclusion

Throughout this practical, various methods for optimisation were explored, and many interesting patterns were observed. Luckily, it was possible for the C code to be optimised to the point where it ran faster than the Python code. In conclusion, the program ran the fastest when the number of threads were set to 32, the compiler flags O2, O3, Ofast and funroll-loops were added and the bitwidth was changed to double.

VI. Github repository link

The following link provides a path to the Github repository that was used to store information for this practical:
https://github.com/tshaihan/EEE3096S.git