Benchmarking
A Python and C/C++ Comparison

Steven Thomi - *THMSTE021*

***Abstract*—The performance of C and C++ algorithms is studied under a variant of optimizations and compared to a golden standard written in Python.**

I. Introduction

THE objective of this research is to investigate how programming languages compare with regard to performance and execution speeds and how bit-width manipulation, multi-threading, and compiler optimization techniques improve overall software performance.
The core findings of my research are that programs written in C and C++ programming languages are easier to optimize and often offer faster execution speeds over their higher-level alternatives.

II. Methodology

A. Hardware

As I have not yet received the Raspberry Pi, these experiments were conducted on my laptop computer. I look forward to replicating these experiments on the Raspberry Pi hardware, once I receive it.

B. Implementation

1) Each "Average Run" was calculated from a set of three values averaged together to give a rough bearing of the speed of the process.
2) I compiled my Python code by calling the Python file through terminal.
3) I compiled my C code using the makefile. I; however, made a few changes to suit my hardware:
   - I changed the compiler to the local gcc compiler.
   - I compared the speed and accuracy performance of the Timer.cpp tic() and toc() functions to the inbuilt C timing library (time.h).
   - I also removed the space formatting – I prefer tabs.

III. Results

A. Python "Golden Measure" Code Run

TABLE I
"Golden Measure" Table

Bit-width	Average Run (ms)
Float (32 bits)	18.660

TABLE II
Non-Optimized Table

Bit-width	Average Run (ms)	Speedup (ms/ms)
Float (32 bits)	0.751	x24.85
Float (64 bits)	1.231	x15.16

B. C Code Run

Non-Optimized Code
C programs have a faster execution time than Python programs; even when the precision in the C program is doubled, C is still significantly faster than the Python algorithm.
The difference in execution time is also as a result of C programs being much closer to machine language than Python programs, which are closer to human language.
In addition, Python is an interpreted language (a line-by-line walk-through of the code is run, detailed by multiple instructions) rather than C which is a compiled language (the entire file is run at once with a few instructions guiding the effort). Therefore, executing Python code requires a larger set of instructions than C, making Python slower than C to execute.

C. Multi-threaded Optimization Run

TABLE III
Multi-threaded Optimization Table

Number of Threads	Average Run (ms)	Speedup (ms/ms)
1 thread	0.849	x21.98
2 threads	0.509	x36.66
4 threads	0.336	x55.54
8 threads	0.314	x59.43
16 threads	0.311	x60.00
32 threads	0.168	x111.1

My laptop is a dual core Intel Core i5. The benchmark runs faster every time a higher-threaded program is run.
Threading enables parallel processing of the algorithm, therefore facilitating a faster processing pipeline and an increase in the Instructions per Cycle (IPC).

From the results, I can infer the following:

- An increase in processing units (threads) results in an increase in processing speed.
- The increase in processing speed plateaus after surpassing the hardware's core count – requiring a significant increase in threads i.e. from 16 threads to 32 threads, to obtain a noteworthy increase in execution speed.

D. Compiler-Flags Optimization Run

TABLE IV
COMPILER-FLAGS OPTIMIZATION TABLE

Compiler Flags	Average Run (ms)	Speedup (ms/ms)
-O0	0.836	x22.32
-O1	0.608	x30.69
-O2	0.602	x31.00
-O3	0.480	x38.88
-Ofast	0.506	x36.88
-Os	0.499	x37.39
-Og	1.213	x15.38
-O1-funroll-loops	0.732	x25.49
-O1-funroll-loops	0.516	x36.16
-O2-funroll-loops	0.985	x18.94
-O3-funroll-loops	0.497	x37.55
-Ofast-funroll-loops	0.510	x36.59
-Os-funroll-loops	0.678	x27.52
-Og-funroll-loops	0.943	x19.79

-O3 and -Ofast-funroll-loops provided the best speedup times.
-O3 is designed to increase the speed of program execution, making trade-offs on executable size.
-Ofast -funroll-loops combines both the ability to break a few rules to go faster and that of unrolling loops into assembly language to enhance program speed.
These compiler flags were the ones expected to give the program the greatest boost in speed.

E. Bit-Widths Optimization Run

TABLE V
SPEED TEST TABLE

Bit-width	Number of Bits	Average Run (ms)	Speedup (ms/ms)
double	64	1.128	x16.54
float	32	0.751	x24.85
__fp16	16	4.448	x4.195

1) Speed Test: An increase in program precision results in a decrease in program speed.

Note: On ARM targets, GCC supports half-precision (16-bit) floating point via the __fp16 type. In cases where hardware support is not specified, GCC implements conversions between __fp16 and float values as library calls.[1]

In the case of __fp16, a fast performance is expected but as I am running the code on an Intel processor (not ARM based), my __fp16 declarations are undergoing a type-cast to another floating-point scheme. On the ARM processors, this should be the fastest of the three alternatives due to its low precision.

2) Accuracy Test: The "long double" bit-width has the highest precision in a floating-point type. It will therefore be used as the "golden measure" with respect to accuracy.

TABLE VI
ACCURACY TEST TABLE

Bit-width	Number of Bits	carrier[90]	% deviation
long double	80	1.90211303259031	-
double	64	1.90211303259031	0.00000000
float	32	1.90211308002472	-0.0000025
__fp16	16	1.90234375000000	-0.0121295

A randomly selected value carrier[90] will also be assessed in this evaluation; its deviation characteristic from the true value is assumed to hold throughout the data sets.

From the experiment, we can infer that an increase in the number of bits results in an increase in the accuracy of the result (least deviation from true value).

F. Best Combination

The following is the best combination of bit-width and compiler flags to give you the best possible speed up over your "golden measure" implementation.

TABLE VII
BEST COMBINATION TABLE

Combination	Average Run (ms)	Speedup (ms/ms)
32 threads, -O3, float	0.140	x133.29

G. Further Tests

I test that the results of the Python code - our golden measure - and our fully optimized code produce the same results by using an accumulator. I summed up result[i] values for all i for both the golden measure and our fully optimized C code.
I observed the similarity of the result in an analysis of whether the two code variants produce the same results.

The C code is accurate to two decimal places to the Python code.

TABLE VIII
FURTHER TESTS TABLE

Code	Sum
C	-0.00152588635683
Python	-0.00000000004434

IV. CONCLUSION

Through the findings of the experiment performed, I can conclude the following:

- The program ran fastest when it was programmed in a compiled 3rd Generation language - C/C++.
- The program ran fastest when it used low-precision floating-point types, despite having a reduced accuracy.
- The program ran fastest when it implemented multi-threading programming.

- The program ran fastest when it implemented speed enhancing compiler optimizations, despite resulting in an increase in the executable size.

REFERENCES

[1] GCC, the GNU Compiler Collection,(n.d). *Half-Precision Floating Point* [Online]. Available at: https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Half_002dPrecision.html. (Accessed: 25 September 2020)