Benchmarking with different programming language and optimisation using different bit widths, compiler flags and multithreading

Mazisi Dungelo[‡] and Tyron Nyambe [§]
EEE3096F Practical 1 Report
University of Cape Town
South Africa
[†]DNGMAZ001 [‡]NYMTYR001

***Abstract*—This document contains observations and results from experiments carried out to determine the effect of various factors (language, threads, bit widths, compiler flags) affecting the performance in Embeded System taking into grave consideration the speedup associated with each.**

I. Introduction

This practical shows how using different programming languages can impact the performance of the program. Also shows the importance of C or C for embedded systems. Here it is going to be shown by running same program written in both Python and C and then compare the speed up.

The other aim is to optimise the C code by multi-threading, different bit widths and compiler flags. Using different compile flags, bit widths(2 bits, 4 bits,etc..) and varying number of threads in a program to find a best combination to optimized the C code. This will thus prove the importance of C in embedded systems.

II. Methodology

The following are the methods used to reach the aim of the experiment, including the software and hardware

A. Hardware

Qemu (Quick emulator) was used to run the programs written in Python and C. It is capable of virtualisation and emulation. It emulates the hardware or software system inside a different host system.

B. Implementation

Qemu was installed in a Linux operating environment. Two versions of the same program were written with Python and C languages. The Python executed code results i.e the averaged time after cache warming up was taken as the Golden Measure for this experiment. Measured and recorded for 3 trials and averaged for both programs in which the average values were used to calculate the speedup. Optimisation was done through changing the number of threads(parallelization) for the threaded version of the C code, changing compiler flags and bit widths to get different speed ups.

C. Experiment Procedure

- Both Python and C (unthreaded) were compiled and ran.Execution times were averaged and used to calculate the speedup.
- C (threaded)-first optimization was changing number of threads from 2-4-8-16 and lastly 32 threads. The average execution times for different threads were used to calculate the speedup in accordance to the golden measure as mentioned above.
- Second optimization involved changing the compiler flags in the makefile to -O0, -O1, -O2, -O3, -Ofast, -Os, -Og and -funroll-loops. The best performing compiler flag combination was -Os-funroll-loops after multiple combinations as we were looking to optimize the program and get the best speedup. The number of threads with best speedup was used in determining the best possible combination.
- Third optimization involved changing the bit widths by using double(64-bits), then float(32-bits) and lastly fp16(16-bits) . The bit width giving best speedup performance was determined based on largest speedup in relation to the golden measure
- By combining different compile flags with different number of bit widths(double:64-bits, float:32-bits, fp16:16 bits) to find the best combination.
- The best combination of compile flag and bit widths was combined with varying number of threads to find the best combination giving best speedup.

III. Results

The following are the results from the experiments.

A. Figures

The following shows the execution time for Python (Gold measure) including the execution time and speed-up for C code

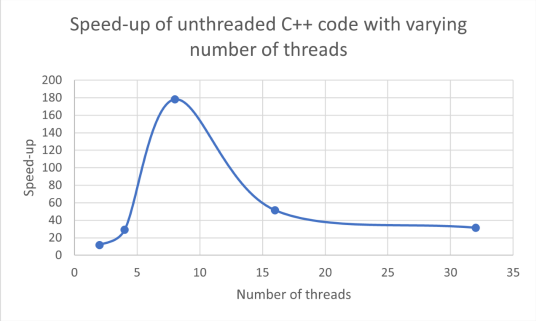

Fig. 1: The speed-up over varying number of threads is increasing

Figure 1 above illustrates how the speed-up increases the more threads are used . This is because more threads virtually means more processors handling work divided among threads running at the same. However, we see that the speed-up decreases with more threads added as shown after the speedup peak in Figure 1. This is because of multi-threading and concurrency issues such as locality issues, data races, load-balancing, synchronization of threads competing with one another, thus speedup decreases.

B. Tables

Table I shows the execution time for Python and C code before optimization

TABLE I: Execution time and speed-up for Python and C(Unthreaded)

Language	Average Time(ms)	Speed-up
Python	496	1
C(unthreaded)	24.02	20.65

Table I proves that C code is has improved performance as shown by the speedup as compared to Python. This is true because Python is high-level language meaning it is abstracted from the essential details of the computer's hardware unlike C which has no or little abstraction allowing direct low level manipulation of registers hence more compact and faster run-time code.

Table II shows the results from the first optimization through varying number of threads in C code.

TABLE II: Execution time and speedup of unthreaded C code with varying number of threads

No.OfThreads	Average Time(ms)	Speed-up
2	42.533	11.69
4	17.0939	29.02
8	2.7845	178.13
16	9.6343	51.48
32	15.74283	31.51

TABLE III: Execution time and speedup of unthreaded C code with different compiler flags

Compile flags	Average Time(ms)	Speed-up
(CFLAGS)	2.7845	178.13
O0	6.6932	74.11
O1	2.6848	184.74
O2	1.6707	296.88
O3	15.1749	32.68186
Ofast	2.6558	186.76
Os	3.2965	150.46
Og	5.0875	97.49
-O2 -funroll-loops	5.3427	92.84

TABLE IV: Execution time and speed-up of unthreaded C code in different compile flags and bit widths

	Average Time(ms)					
C flags	double(64-bits)	Speed-up	float(32-bits)	Speed-up	__fp16	Speed
CFLAGS	5.84	85.93	2.83	175.27	2.76	179.7
O0	7.65	64.84	6.54	75.84	5.28	93.94
O1	13	38.15	3.02	164.23	3.98	124.6
O2	3.85	128.83	1.55	320	1.23	403.2
O3	24.43	20.30	13.18	37.63	10.34	47.97
Ofast	2.87	172.82	3.21	154.52	2.45	202.4
Os	9.23	53.74	4.2	118.10	3.4	145.8
Og	3.28	151.22	4.88	101.64	5.12	96.87
-O2 -funroll -loops	4.58	108.30	4.96	100	2.67	185.7

The bit width affects the space memory, this also affect the speed of the process. If processing values from the memory stored in high bit width the program will obviously be slower. This is seen in the above results that double: 64 bits is slower than float:32 bit. However for 16 bit is faster but sometimes slower because accessing 16 bit in memory does not take much processing, the problem is when the number is much bigger than 16 bits the processor gets slower when bit widths used is 16 bit.

TABLE V: Execution time and speed up from varying number of threads under -O2 compiler flag and 32 bit width

-O2 and float (32 bits)		
No.OfThreads	Average time(ms)	Speed-up
2	49.84	9.95
4	22.27	22.27
8	9.04	54.87
16	14.55	34.09
32	18.64	26.61

IV. Conclusion

By varying the number of threads, bit widths and compiler flags we found the best speedup when: Number of threads=8 Compiler flag=O2 Bit width=fp16 giving us a speedup of 403.25 for C threaded program. For C unthreaded program the best speedup was obtained when: compiler flags=O2 Bit width=float(32) resulting in a speedup of 296.88. Drawing

from this, we can conclude that by multithreading our program we can yield large speedups in our computations as compared to our golden measure. This therefore implies through multithreading we can improve our system performance and by using C language as compared to python can substantially increase our speedup due to its low level direct manipulation with the Embeded Systems Hardware.

REFERENCES